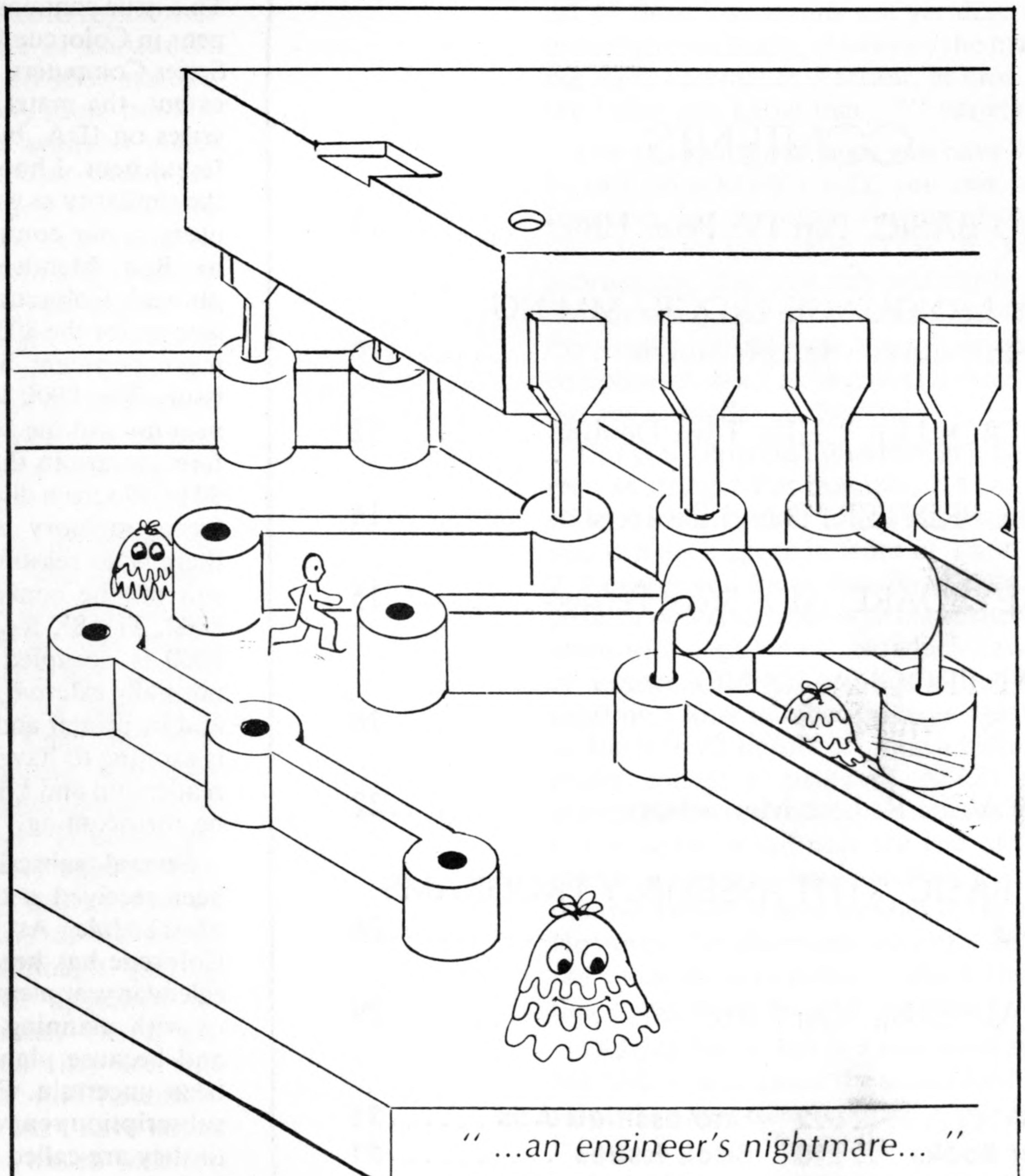
COLORCUE

VOLUME VI
NUMBER 4

A BI-MONTHLY PUBLICATION BY AND FOR INTECOLOR AND COMPUCOLOR USERS



"...an engineer's nightmare..."

W. S. Whilly on the rampage again....

Compiling Basic

ANIMATION

First 8000 Article!!!

Tiny-PASCAL

Calling Assembly Routines from BASIC

JULY/AUGUST 1984

Colorcue

VOLUME VI, NUMBER 4 JULY/AUGUST 1984

CONTENTS

EDITOR: JOSEPH NORRIS COMPUSERVE: 71106, 1302

COLORCUE is published bi-monthly. Subscription rates are US$18/year in the U.S., Canada, and Mexico (via First Class mail), and US$30 elsewhere (via Air Mail). All editorial and subscription correspondence should be addressed to COLORCUE, 19 West Second Street, Moorestown, NJ 08057, USA. (609-234-8117) Every article in COLORCUE is checked for accuracy to the best of our ability but is not guaranteed to be error free.

"8000 SPOKEN HERE!!!!!"

This issue contains the first article to appear in Colorcue for the Intecolor 8000 Series Computers. It duplicates, to some extent, the material in W. S. Whilly's series on IDA, but is of a slightly different bent. I hope readers will forgive the similarity as we push to include 8000 users in our community. Many thanks to Bob Mendelson for taking the plunge! Colorcue has received ROM listings for the 8000 and will publish the more pertinent addresses in the next issue. The 8000 is very like the Compucolor and the 3651. The primary differences are in the 24K user RAM, an 80 by 48 screen display, and a totally different memory mapping. In general, there is no reason why CCII programs can not be converted for use on the 8000. The 8K RAM not present in the 8000 is occupied by Command files, normally external in the 3651 and CCII, and by printer and light pen routines. It is exciting to have this extension to our readership and I hope more articles will be forthcoming.

Several subscription renewals have been received at the office, well ahead of schedule. As you probably know, Colorcue has been accepting only full calendar year memberships, both to help us with planning on a six-issue basis, and because plans for next year have been uncertain. Please do not send in subscription renewals for next year until they are called for in Colorcue, that is, unless you owe more money to complete the 1984 edition. We will be making plans for next year during the remainder of the summer and announce them in the Sept/Oct issue.

As the Sourcebook materials continue, you will notice a dearth of material on hardware. PPI in Australia and Tom Devlin, in Michigan, are virtually the only vendors selling a selection of hardware materials for the CCII. Tom Devlin continues to offer his RAM card and Analog Protector circuit. Ben Barlow is offering a lower case character ROM (See previous issue). I do not know if Frepost Computers still exists.

from the Editor's Desk

Communications with them have been unanswered. I would like to repeat that any materials, software or hardware, that you want to add to your system, should be purchased very soon. It is not likely that they will be available much longer. The Rochester User Group remains the strongest source of software and overall inspiration. Join them to assure their continuation, and to make available to yourself a rich deposit of software in the free library. Colorcue is interested in having reviews of interesting programs in the CHIP library. This is a good time for you to write one on a CHIP program that has been interesting to you.

Since the publication of the ROM tables in Colorcue, we will no longer specify multiple ROM locations in articles, both to save editing time and to encourage you to use the tables. It is appropriate that our articles lean more heavily on Assembly Language programming than ever before. Assembly programming is the highest point in the Compucolor experience. It offers the greatest power. It is very enjoyable to work on, and it brings one as close as possible to the wonders of the Compucolor computer. Although 'getting started' articles have appeared in FORUM and Colorcue, there are readers who write that they are still puzzled and 'all at sea' with Assembly Language procedures. As I have noted previously, one almost has to wait for the 'light' to turn on. It will not turn on without your help, however. If you are among those still in the dark, we invite you to write to us for a personal tutorial in Assembly programming, tailored to your own special needs. Please state in your letter what you have already done, your analysis of your present problems, and what you think might be helpful. We will try to respond accordingly, even if you don't know where to start. It isn't difficult, truly, and the rewards are tremendous. You owe it to yourself to accept the challenge. □

John.

COMPILING BASIC

Part IV.

Peter Hiner
11 Penny Croft
Harpenden
Herts, AL5 2PD
ENGLAND

For this article, the last in this series, I am left with a long list of Basic commands not yet described and some other miscellaneous items. However, the majority of the remaining Basic commands warrant, at most, a brief comment of the "Did you know that...?" variety.

Did you know that once you have defined a function (as in DEF FNA(X)=X+Y-Z), you can subsequently use that function in defining further functions (as in DEF FNB(M)=FNA(1)+N)? This is rather like nesting subroutines, and you can add further levels of complication by using FNB in the definition of yet another function. Personally, I find even a single level of FN functions too complicated, but I came across nested functions while testing out my compiler on Startrek.

Did you know that numbers in a DATA statement can be read as strings? For example, DATA 1,2,3 could be followed by READ A,B$,C. If you subsequently PRINT A; B$; C, you will get a space in front of 1 and 3 but not in front of 2. I expect you knew that Basic puts the space in front of positive numerical values to make them line up neatly in columns with negative values (which have a minus sign instead of a space.) But did you know that Basic carefully avoids splitting a long number (such as 123456) between two lines on the screen? Before printing a numerical value, the interpreter creates a string of ASCII characters (in string manipulation space) and counts the number of characters. If the cursor is too near the end of a line, the interpreter inserts a carriage return before it starts printing out the number (which it now handles in the same way as a string.) However, the statement A$="123456" : PRINT A$, would not cause the interpreter to check the cursor position, and this string could be split between two lines.

Did you know that in a statement such as PRINT "DEAR ME";N$, you can omit the semicolon? I learned this the hard way when someone reported a bug in an early version of FASBAS.

Did you know that the statement PRINT SPC(X) does not cause a carriage return and line feed, even though it is not followed by a semicolon? The same applies to PRINT TAB(X). Unfortunately, I have only just had this pointed out to me, so there is still a bug lurking in my compiler, which can be avoided by putting a semicolon after these statements.

While on this subject, I should advise you of the other known bug. If you have a PRINT statement which contains a "cursor down" character, it will compile without problem, but, during the subsequent assembly operation, the "cursor down" character will look like an end-of-line marker, causing chaos and an error message. This problem can be overcome by replacing the "cursor down" character with

CHR$(10), although that will be marginally slower. To maximize speed, you could be brave and edit the intermediate version of a compiled program before assembly, but I doubt that you would notice the difference in speed.

To avoid having this article become a jumble of miscellaneous items, I will move on to the subject of memory space, beginning with a map to compare the allocation of blocks of memory for Basic and compiled programs. (See Fig 1.)

The manual supplied wih *FASBAS* describes the difficulty in defining (for either Basic or compiled programs) the limits of the "spare space" block, squeezed between blocks allocated forward from 829AH and blocks allocated backward from the end of memory. The "spare space" block will vary in size, both from program to program and during the course of running a program, so it is a dangerous practice to use it for machine code routines. The safe place for these is in space reserved at the end of memory, and this space can be reserved by starting with a line like this:

```
0 POKE 32940,a : POKE 32941,b : CLEAR 50
```

FIG 1. Relative memory allocations

ADDR	BASIC	COMPILED
8000H	--- FCS Parameters ---	
8200H	--- Basic parameters ---	
829AH	Basic program including DATA	Run-time library Data, Compiled program Variables & constants One dimensional numerical arrays
????	Variables and string pointers	String pointers
????	All types of array and File buffers	String arrays File buffers Multi-dimensional numerical arrays
????	--- Spare space ---	
????	--- Space for stack ---	
????	--- String space ---	
????	Space can be reserved here for machine code routines	
END OF RAM		

The values for a and b can be found from Fig 2. The POKE statements set a limit on the size of memory available to the interpreter, and the CLEAR statement defines the size of string manipulation space to be allocated (counting backward from the new limit of available memory.) The stack space is automatically moved further down in memory at the same time.

If you have a program which runs out of memory space when using FILE routines, you will have to consider reducing the size or the number of file buffers. The Compucolor Basic Manual gives all the information you need, but requires very careful reading to appreciate the implications of the decisions you may make concerning blocking factor and number of file buffers allocated.

The fast and easy way is to allocate enough buffers to enable the whole file to be stored in RAM at once, but I will assume, now, that you have run out of memory space. If you had allocated more than one buffer in the FILE "R" statement, you can reduce the number of buffers without any problem, except that the number of file accesses may be increased. If you had only allocated one buffer, you can use the option allowed by the FILE "R" statement to override the blocking factor. This can be done without changing the file itself, provided that everything (including your new blocking factor) fits exactly into a pattern of multiples of 128 bytes, or else you will be in trouble. If this suggestion leaves you confused, then do not attempt to implement it.[1]

If you are starting a new file, then you can get things right in the first place. The number of bytes per record multiplied by the blocking factor (number of records per block) determines the size of one file buffer (we can ignore the additional bytes used for housekeeping.) If this resulting number is not an exact multiple fof 128, then you will waste space both on the disk and in RAM.

Let us take as an example a file consisting of 256 records, each containing 32 bytes, and then consider the effects of varying the blocking factor and the number of file buffers allocated. To read the entire file at once from disk would require 8K of memory space, and it would make no difference whether we chose a blocking factor of 64 (and therefore allocated one file buffer of 8K bytes) or chose a blocking factor of 4 (and therefore allocated 64 file buffers of 128 bytes each). Other combinations between these extremes would also give the same result.

If, however, we could only afford 512 bytes of memory space for the buffers, then the maximum value of blocking factor we could use would be 16, and we would then allocate 1 buffer. This would cause 16 records to be read every time the disk is accessed, and would be the best arrangement (within this memory limitation) for sequential file access, or for most forms of random access.

To see why it is normally best to make the blocking factor as large as possible and to allocate only one file buffer, let us consider what would happen if, in the above example, we chose a blocking factor of 4 and allocated 4 file buffers of 128 bytes each. The first file access would cause 16 records to be read from disk (just as previously) and these

would fill all 4 file buffers. But any subsequent access to a part of the file not already in memory would cause only four more records to be read from disk. These four records would be put in the "least recently used" file buffer (overwriting the previous contents.) You can see that this is likely to result in nearly 4 times as many disk read operations, and will therefore be much slower. The only case in which you would benefit from multiple file buffers is when you want to retain part of the file (such as an index) in memory all of the time, and to achieve this you might have to insert dummy GET statements to assure that those buffers you wish to retain do not become the "least recently used."

FIG 2. Values for Reserved Memory Space.

Number of bytes to reserve	16K Memory (a)	16K Memory (b)	32K Memory (a)	32K Memory (b)
128	127	191	127	255
256	255	190	255	254
512	255	189	255	253

In one way or another, FILE statements gave me quite a lot of trouble while writing FASBAS. For a start, I had never been a heavy user of FILE statements in Basic programs, so I had to learn how they were meant to work in Basic before I could even comtemplate compiling them. I could see that FILE routines might spend a lot of time accessing the disk, which would not offer any chance for speed improvement. So I did not apply much effort to them initially, and I only included the minimum facilities in *FASBAS* v12.20. I still managed to get some bits wrong, and while correcting these bugs for v12.21, I decided to try to provide a complete implementation of all the FILE statements, including the obscure FILE "A" command. (Does anybody ever use it?)

The most difficult FILE statement was FILE "T", which provides error trapping. The theory is that if you include in your Basic program a statement like FILE "T",1000, the interpreter will jump to line 1000 instead of giving an error message, if at any time it finds an error while executing a FILE command. The trap facility is turned off again by declaring FILE "T", without a line number. The problem confronting me was that the error trap routine would try to find and interpret line 1000, and I could not make the interpreter give control back to the compiled program. The file routines are long and complex, containing many check points which might cause the program to jump into the error trap routine. So I could not put an alternative error trap routine in the run-time library unless I was prepared to rewrite all the file routines as well.

I came to the conclusion that I would have to include a bit of Basic program to satisfy the interpreter when an error was trapped. This bit of Basic program would contain an escape mechanism to give control back to the compiled program. Since the interpreter would start from address 829AH in its search for the required line, the compiled program would have to start with one or more lines of Basic before the machine code. The solution I eventually chose was related to the solution to another problem (chaining compiled programs), but I will try to keep them separate for the moment.

First of all, I made life a bit easier by determining that the error trap routine would always redirect the interpreter to the same line number (I chose line 2 for reasons which will be apparent later.) The compiled program would start with three lines (0, 1 and 2) of Basic, and line 2 would look like this:

```
2 POKE 33216,242 : POKE 33217,130 : PLOT 27,94
```

The POKE statements put the value 82F2H into memory at address 81C0H (which is the location of the jump vector for [ESC] [USER]. PLOT 27,94 is equivalent to keying in [ESC] [USER], and this will result in the program being vectored to 82F2H, which is the start of the routine for error trapping. So we have achieved the first part of our objective, by escaping from the interpreter routines to our own machine code routine.

The error trap routine contains an instruction JMPwxyz, where 'wxyz' represents an address which has previously been inserted during the execution of a FILE "T" statement (in our example the address inserted would be the location of the compiled version of line 1000.) So by a devious route, the program will eventually reach the right place.

Now we can look at the first two lines of Basic program, which are included to provide a mechanism to chaining compiled programs.

```
0 REM (followed by what appears as garbage)
1 POKE 33215,195 : POKE 33216,154 :
  POKE 33217,130 : PLOT 27,94
```

The POKE statements put the assembly language instruction JMP 829AH into the user escape location, and the PLOT 27,94 causes user escape to be activated. So if the Basic interpreter were to start reading what looks like the beginning of a nor-

Fig 3. Beginning compiled code lines.

Address	Object Code	Assembly	Meaning in Basic
829AH	A3H	ANA E	Address of next
829BH	82H	ADD D	line of Basic
829CH	00H	NOP	Basic line
829DH	00H	NOP	number 0
829EH	8EH	ADC M	REM
829FH	C3H,FDH,82H	JMP 82FDH	Rubbish
82A2H	00H	NOP	Basic end-of-line marker

mal Basic program, it would start at line 0, find a REM statement, and ignore the rubbish in the rest of the line. Then it would execute line 1 and the user escape function ould cause the processor to jump out of the interpreter routine and go back to address 829AH.

This time, the program would no longer be under the control of the interpreter and therefore the code starting at address 829AH would take on an entirely different meaning. To explain this, I have listed the address, object code and assembly language version of the first few bytes in Fig 3.

The code in 829AH to 829EH (which is really the Basic linking address, line number, and REM token) does nothing useful when taken to be machine code instructions, but it does no harm either. The JMP instruction then directs the program to the correct address after the rest of the Basic lines and the error trap routine.

Fig 4. Lines for pseudo-Basic program.

```
        ORG  829AH

        DB   0A3H,82H,0,0,8EH
        JMP  BEGIN
        DB   0,0CFH,82H,1,0
        DB   95H,'33215,195:'
        DB   95H,'33216,154:'
        DB   95H,'33217,130:'
        DB   92H,'27,94',0,0,0

BEGIN:  ; The rest of your program here.
```

Now we have a compiled program which can be run as a PRG type program under FCS control, or can be accessed through the Basic interpreter. So if we change the file type from PRG to BAS in the directory, we can load and run it just like a normal Basic program. We can freely chain together any combination of Basic and pseudo-Basic (compiled) programs using LOAD ....: RUN statements.

If you want to make your own assembly language programs look like Basic programs, you can include something like the Basic lines 0 and 1 at the beginning. (They must load to 829AH.) You can change the file type in the directory from PRG to BAS using the FCS RENAME instruction and then treat the program as if it were in Basic. One possible application of this would be to name a program as MENU, so that it can be run from the AUTO key.

For your convenience I give a listing in assembly language of the instructions you would need at the start of a pseudo-Basic program in Fig 4. This concludes my series of articles on compiling Basic. I hope you have found this ramble around the subject interesting and, in places, useful.

[FASBAS is available from the author for $25US. Ed]

1. See also COLORCUE, VOL VI, No 2, pps 26-28. Ed.

Assembly Language Programming

Part XV. Joseph Norris

Why are we so fascinated by animation? I wouldn't want it told that in spite of a very proper upbringing and a lifetime of rather sophisticated intellectual pursuits, I can be hooked for hours at the terminal trying to outwit a dumb figure that I know, perfectly well, is programmed to do me in, everytime. So, welcome to the human race!

This article has been requested more often than any other, yet you already have a good feeling for the construction of animation in assembly language. The fact that much of what we do here will seem obvious and elementary will be evidence of that. Since our purpose is to explore possibilities, rather than create a finished program, we will look at some techniques, carry them somewhat into a meaningful area, then cruelly leave you on your own, with your imagination and an inspirational screen display to use as a "springboard" for further play.

Animation on the CCII/3650/8000 computer can be achieved through the family of PLOT functions enabled in the system ROM, and, more satisfactorily, through the direct use of screen memory. We will begin by examining a brief example of animation using the PLOT functions.

In BASIC, we can construct and plot a rectangular "animaton" with a single line:

```
105 PLOT 3,30,15,2,110,111,3,30,16,2,100,109,255
```

This line sets the cursor at x = 30, y = 15; enters the character plot mode ("2") and prints the 1st and 4th quadrant "corner" figures where they belong.

To add dignity to the rectangle, we can enter the blind cursor mode by preceding the x,y coordinates with a number higher than the largest valid x coordinate (greater than "63"). I have chosen "82" because it satisfies a "bug" in early software versions. (See the Instruction Manual for a description of the blind cursor mode.)

```
105 PLOT 3,82,30,15,2,110,111,3,82,30,16,2,100,109,255
```

If we change the values of x and y in line 105 then we can make the animaton "move" through the screen area. This is done easily in Basic by placing variables X and Y in the string, and replotting the string with changing values of X and Y:

```
105 PLOT 3,82,X,Y,2,110,111,3,82,X,Y+1,2,100,109,255
```

The illusion of motion of an animaton requires that we erase the previous position of the animaton before plotting its new location. To erase the rectangle, we can construct another line that is identical to Line 105, except that it will print "spaces" ("32") over the graphics characters, hence "erasing" them:

```
115 PLOT 3,82,X,Y,2,32,32,3,82,X,Y+1,2,32,32,255
```

animation

These two lines, executed in repeating succession, produce a "blinking" character plot on the screen. We may plant these same lines, virtually untouched, into an assembly routine as labelled DB strings. The only difference is the addition of "239" at the end of each string, so OSTR can be used to print them:

```
MAIN:  LXI   H,GRAPH  ;Point to string
       CALL  OSTR     ; and print it
       LXI   H,CLEAR  ;Point to erase string
       CALL  OSTR     ; and print it
       JMP   MAIN     ;Do it all again!

GRAPH: DB 3,82,30,15,2,110,111,3,83,30,16,2,100,109,239

CLEAR: DB 3,82,30,15,2,32,32,3,82,30,16,2,32,32,239
```

To change the plotting position of the rectangle, we need only alter that portion of the memory contents of these two strings which determine the x and y plotting values. For GRAPH:, these are the third and tenth bytes for the x values (GRAPH + 2,GRAPH + 9), and the fourth and eleventh bytes for the y values (GRAPH + 3, GRAPH + 10); similarly for CLEAR:.

Before we can experiment with this animaton, we need a skeletal program to perform the plotting and get some interfacing from the "joystick" or keypad. I call, again, on David Suits's keyboard input routine, from the June/July 1982 Colorcue. We need only those portions that will "get" a character press and place it in the accumulator for further processing. (Please refer to that article for explanation of this portion of the source code.) Specifically we will use the routines labelled "TEST", "GTCHA", and "CHRINT."

Our need for a skeletal program is filled by GRAPH.SRC (see Listing I). The comments are rather thorough, but I offer the following additional explanations:

Keyboard Assignments. I have chosen to use the numeric keypad for operator interfacing with the program, following the convention of "CHOMP", with "4" and "6" meaning "left" and "right", and "8" and "2" meaning "up" and "down." My joystick is connected to these keys. If you have a joystick assigned to the "arrow" keys, then make the appropriate conversions in GRAPH.SRC to accommodate them, by changing the CPI values in MAIN. If you have no joystick, then assign any keys that are comfortable for you. One such arrangement that works well is to use "N" and "M" for left and right, and "D" and "C" for up and down—using two hands for control.

The ten lines of code at MAIN control the proceedings. Their purpose is to intercept a keyboard input and branch to the appropriate subroutine for action. Key in GRAPH.SRC and assemble it to an appropriate origin. What you see when you RUN the program doesn't seem very exciting. A stilted rectangle moves inelegantly in four directions within the screen boundaries we have imposed. We will take some steps, however, to transform this into a rather interesting animaton.

Suppose we bypass the GTCHA routine, and cause the program to operate at full speed, maintaining its previous branching action until a new keypress is stored in KBCHAR. Change the source code at MAIN:

Replace MAIN: CALL GTCHA with MAIN: LDA KBCHAR.

With this change, the last keypress will remain in effect until a different keypress is stored in KBCHAR ("typematic" action, so to speak.) If you assemble this code you will now have a more challenging animaton on your hands, zipping from side to side and up to down , "out of control." But you have some important information in this demonstration. You now know how fast the PLOT structure can move things on the screen!

Let's slow things down a bit. Replace the single line at MAIN with these two lines:

```
MAIN:  CALL  WAIT    ;Pause a bit
       LDA   KBCHAR  ;Get last keypress
```

Add the WAIT subroutine in Listing 2. following the PRINT subroutine. This handy timer has a very large range of delays, and you will enjoy placing different numbers in the B register (by changing the source code and reassembling). You now have reasonable control of the rectangle. Notice that as the speed increases (as the value of B goes down), the lowly rectangle takes on a more interesting aspect. In a single additional step, we can create a "useful" game. We will paint the background blue, and permit the rectangle to trace a path through the screen as it moves. Change the DB string CLR to read as follows (adding a second line):

```
CLR:  DB    6,36,12,27,24,15,30,6,2,3,0,30,11
      DB    3,0,31,11,3,0,31,'SCORE: ',239
```

Plot 6,36 gives us a blue background (other codes will do that as well, of course) and we have "erased" the blue on lines 30 and 31 for future use as a scoring area. In mid-string, we have changed to green on black, so our rectangle will plot in those colors through the blue background we just layed out.

Now assemble and run GRAPH.PRG. Now see how "animated" our rectangle has become, "eating" its way around the screen under joystick or keypad control. Try to make a continuous path around the screen without intersecting any previous path; then test your skill at retracing it without going off the path (see FIG 1). (It isn't easy if B = 32 or less.) If you get tired, pressing the "fire" button (or any non-defined key) will stop the rectangle in its tracks. (We are touching on "DIG-DUG" territory.)

Making a useful "game" of this skeleton program isn't difficult. Your ideas will be better than mine, but here are some inspirational thoughts. Suppose we began the game with a highest possible score of 9999 and B = 16. A subroutine called at the beginning of MAIN decrements the score by one. Your goal is to gobble up all the accessible blue area before the score reaches 0. After displaying the final score of the first game, the program recycles with a still lower value in the B register, getting faster and faster each time it's played.

Another interesting set of refinements comes from a subroutine that can tell if the animaton is about to travel into a previously "erased" area or not. This permits scoring in a game that wants you to retrace a previously etched pathway. Such a subroutine can be derived by testing the CCI character in the plot blocks to be written to next.

With the ability to test plot blocks for their CCI content, we can also "plant" barriers to the animaton in a previously etched pathway, requiring a change of motion, tracing an alternate pathway, all with additional score reductions.

We will need a counting routine for the score as well. These are some things you can work on until next time, and we will then examine the use of direct access to screen memory as a technique for animation. □

Listing 1.

```
;GRAPH: AN ASSEMBLY ANIMATION PRIMER

;               JULY 25, 1984 JHN
;               Turn lower case off, caps lock on

;               Group Equates together here .....

;               Use your ROM tables to get v6.78
;               addresses. ISC 8000 users see end
;               of listing for instructions.

                OSTR    EQU     182AH   ;v8.79 &
                KBCHAR  EQU     81FEH   ; v9.80

;               Set origin for ESC T ............

                ORG     8200H

;SETUP..Set initial conditions ..........

BEGIN:  LXI     H,0             ;Clear HL
        DAD     SP              ;Add SP to HL
        SHLD    FCSSP           ;Store old SP
        LXI     SP,STACK        ;Add new SP

        MVI     A,0C3H          ;Setup for CHRINT
        STA     81C5H           ; See David Suits
        LXI     H,CHRINT        ; input routine
        SHLD    81C6H           ; for details
        MVI     A,1FH           ;
        STA     81DFH           ;

        LXI     H,CLR           ;Clear screen etc
        CALL    OSTR

        JMP     PRINT           ;Draw initial box

;MAIN PROGRAM...........................

MAIN:   CALL    GTCHA           ;Get key press
        CPI     '6'             ;Move right?
        JZ      XINR            ;Jmp right routine
        CPI     '4'             ;Move Left?
        JZ      XDCR            ;Jmp left routine
        CPI     '8'             ;Move Up?
        JZ      YDCR            ;Jmp up routine
        CPI     '2'             ;Move Down?
        JZ      YINR            ;Jmp down routine
        JMP     MAIN            ;Invalid input

;SUBROUTINES............................

CHRINT: PUSH    PSW             ;Suits input routine
        XRA     A               ; See his article
        STA     81FFH           ;
        POP     PSW             ;
        RET

GTCHA:  XRA     A               ;Get keyboard char.
        STA     KBCHAR          ; See David Suits
GTCH1:  LDA     KBCHAR          ;
        ORA     A               ;
        JZ      GTCH1           ;
        RET                     ;

XINR:   LDA     GRAPH+2         ;Move right routine
        CPI     60              ;Right limit?
        JZ      MAIN            ;Yes. Don't move
        CALL    PREP            ;Erase old box
        LDA     GRAPH+2         ;get old x value
        INR     A               ; & increase by 1
        JMP     DOX             ;Store new x value
```

```
XDCR:   LDA     GRAPH+2 ;Move left routine
        CPI     2       ;Left limit?
        JZ      MAIN    ;Yes. Don't move
        CALL    PREP    ;Erase old box
        LDA     GRAPH+2 ;Get old x value
        DCR     A       ; & decrease by 1
        JMP     DOX     ;Store new x value

YINR:   LDA     GRAPH+3 ;Move down routine
        CPI     27      ;Lower limit?
        JZ      MAIN    ;Yes. Don't move
        CALL    PREP    ;Erase old box
        LDA     GRAPH+3 ;Get old y value
        INR     A       ;Increase by 1
        STA     GRAPH+3 ;Store new value
        STA     CLEAR+3 ; in two places
        LDA     GRAPH+10;Get old y1 value
        INR     A       ;raise by 1
        STA     GRAPH+10;Store it also
        STA     CLEAR+10; in two places
        JMP     PRINT   ;Draw new box

YDCR:   LDA     GRAPH+3 ;Move up routine
        CPI     1       ;Top limit?
        JZ      MAIN    ;Yes. Don't move
        CALL    PREP    ;Erase old box
        LDA     GRAPH+3 ;Get old y value
        DCR     A       ;Reduce it by 1
        STA     GRAPH+3 ;Store it in
        STA     CLEAR+3 ; two places
        LDA     GRAPH+10;Get old y1 value
        DCR     A       ;Decrease by 1
        STA     GRAPH+10;Store also in
        STA     CLEAR+10; two places then
        JMP     PRINT   ;Draw new box

PREP:   LXI     H,CLEAR ;Erase current box
        CALL    OSTR
        RET

DOX:    STA     GRAPH+2 ;Store new x
        STA     GRAPH+9 ; value in
        STA     CLEAR+2 ; these four
        STA     CLEAR+9 ; memory slots
        JMP     PRINT   ;Print new box

PRINT:  LXI     H,GRAPH ;Print box at
        CALL    OSTR    ; new x,y &
        JMP     MAIN    ; go back.

;STRINGS...............................

;Setup string:  BG=BK, FG=GR, Erase page
                ;Page mode, A7Off, Flag On
```

```
CLR:    DB      6,2,12,27,24,15,30,239

;String to print animaton................

        ;       BC  x  y  Chars..

GRAPH:  DB      3,82,30,15,2,110,111

        ;       BC  x  y  Chars..

        DB      3,82,30,16,2,108,109,239

;String to erase animaton................

        ;       BC  x  y  Space

CLEAR:  DB      3,82,30,15,2,32,32

        ;       BC  x  y  Space

        DB      3,82,30,16,2,32,32,239

        ;       Storage for Stack Pointer

;Numerical Storage ....................

FCSSP:  DW      1       ;FCS Stack Pointer

SCORE:  DS      2       ;Score storage

;Stack Allocation. This EQU entry means:
;       'Let the address STACK be 80H
;       bytes further on than this address'
;
;       We do this because the stack works
;       backwards toward 'SCORE' and we
;       want to allow enough room so it
;       won't write into 'SCORE's area.

        STACK   EQU     $+80H

        END     BEGIN

        ;Don't forget CR after 'BEGIN'

;INSTRUCTIONS FOR 8080 USERS:

;OSTR   EQU     0901H
;KBCHAR EQU     9FFEH

;       ORG     A000H
```

Notes On The CRT Controller Chip

Tom Devlin
3809 Airport Road
Waterford, MI 48095

Failure of the CRT controller chip in the CCII is a distressing problem. These parts are becoming very difficult to find. The history of this part in the Compucolor computer is a little complicated because three different parts have been used over the years.

The part first used was the SMC Microsystems CRT-5027. This has been, perhaps, the most widely-used CRT controller chip in the industry. It currently lists for $19.00 in the JDR Microsystems catalog and in the most recent Byte magazines. It is programmable for a variety of screen formats, and so it must be re-programmed each time the computer is powered-up. In the v6.78 systems, the data is stored in the PROM, UA1, to be read (as I/O of all things!) and written into the 5027 by the FCS routine from 3774H to 3794H.

At some time early in the v6.78 production, ISC had SMC mask a version of the 5027 for a 64*32 format specifically for the CCII. This version was given the part number CRT-5027-003, and it eliminated the need for PROM UA1, although socket space for UA1 was left on the logic board for quite some time after the change was made.

It is my understanding that the CRT-5027-003 device is now out of production. On v6.78 systems it is possible to replace it with a standard CRT-5027 and a copy of the original PROM. This PROM is an 82S123 (32*8) programmed as follows:

ADDRESS:	00	01	02	03	04	05	06	07-1F
DATA:	35	97	D3	F9	60	30	F1	FF

(I can supply these PROMs at a cost of $20.00 each.)

When ICS went to v8.79, they moved the set-up parameters into the FCS ROM proper, and thus eliminated the need for UA1. Since they used the masked part, this data was never used, but its presence does make it easy to replace the masked (-003) part with one of the standard programmable versions.

The very last Compucolors built used a CRT-5048-3, instituted about the same time as the REV 4 logic board. It is probably not directly replaceable with the 5027. The 5048 is used on the 3651 so availability should not be a problem.

QTZ

```
100 REM  ** COMPUCOLOR II CHARACTER DISPLAY **
105 REM             by J. Ramsey
110 REM     ....Press <RETURN> to escape....
120      CLEAR : L=1 : C=2 : PLOT 12,3,64,0
130         N=28670 : FOR XX=0 TO 127
140           XX$=RIGHT$("   "+STR$(XX),4)
150            FOR I=1 TO 4 : N=N+2
160       POKE N,ASC(MID$(XX$,1,1)) : NEXT
170    N=N+4 : POKE N,XX : POKE N+1,C : C=C+1
180              IF C>7 THEN C=2
190    N=N+4 : POKE N,XX+128 : POKE N+129,C
200     POKE N+128, XX+128 : POKE N+129,C
210         C=C+1 : IF C>7 THEN C=2
220     L=L+1 : IF L>8 THEN L=1 : N=N+128
230            NEXT : POKE 33278,0
240        IF PEEK(33278)=0 THEN 240
250                    END
```

ADDENDUM

It's time for some special thanks: to Peter Hiner for four extraordinary articles on Basic, representing countless hours of work exploring, programming, writing and giving generously of himself; to Jane and Tom Devlin for performances far above the ordinary, and steady support of this magazine and its staff; to the faithful writers of this Volume - Doug Van Putte, W. S. Whilly, and Rick Taubold - who keep us in good company; and to all of you who continue to fill our office with such good materials.

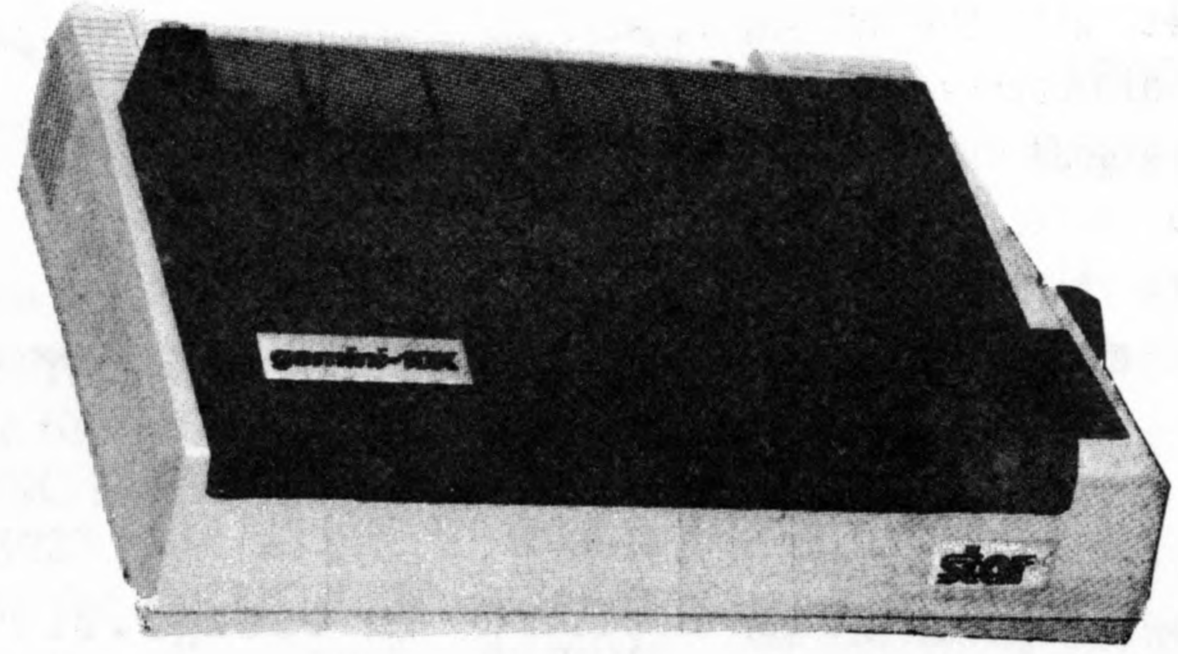
We are still only hearing from a very few. There is room in our pages for much, much more. It's time for your article now! Contributions on animation are especially needed.

So far there have been no entries in the Colorcue Contest announced in the Mar/Apr issue. Does that mean I get to keep the prize money? You might try an entry, you know. If only one person enters..he wins!

We are sorry to lose Tom Andries as a user. He has endured much with a failing CCII. If you haven't tried Tom's Hour Glass graphic from Vol V, Jun/Jul, you're missing a remarkable bit of programming. Try compiling it with FASBAS for some extra pleasure. It is a simple and elegant use of CCII graphics capabilities.

Product Review -

The Gemini 10X Printer



David R. Ricketts
108 Joyce Avenue
Red Bank, TN 37415

It's Thursday of my one week of vacation, it's raining, you want articles, my CCII does not have lower case, my word processor is "COMP-U-WRITER 3.3, I have never written for a magazine before and I can't stand rejection. In spite of all these handicaps, here goes.

Spend $300 for a printer, me! the original tightwad, when this model 35 works great?? Oh! I gotta have a serial board too, even more money, a buffered serial board would be nice, even more money. Oh well, a friend of mine is selling the STAR, GEMINI line. "Take one home, try it out, pay me if you like it and etc.". I did, I did and I did.

I bought the Gemini-10X with the 4K buffered serial board and I have been using it for several months now and, if I can do so without sounding like a commercial, I'll try to share my experience.

DOCUMENTATION: The "USERS MANUAL" (packed in the box) although preliminary, is thorough, even to the point of illustrating the Removal of the Upper Case, Replacement of the Fuse and Replacement of Print Head. Also included are such things as: Parallel Interface Specifications, Connector Signals and Functional Description for Parallel Interface, Block Diagram, Code Chart & etc.

Although a "snow-job" at first, the instructions for set-up became clearer as I began to use them. My "preliminary" manual usage was short lived as my friend (the salesman) provided a much more thorough "USERS MANUAL" which provides sample programs for most of the popular computers (but not the CCII - that's okay 'cause we Compucolor users are accustomed to such). This manual utilizes illustrations liberally, is in an easy to read format, and it's 282 pages are a wealth of information, right down to the Glossary (pages 266 & 267) and the QUICK REFERENCE CHART on the inside back cover. In short, in a field where documentation is so scarce, many manufacturers (and software suppliers) could take a lesson from this book.

Did I mention that I did get the 4010X buffered Serial Interface? Well, it has it's very own USERS MANUAL, although very much like the "preliminary" it too is thorough and includes a schematic diagram, Interface instructions for several computers (not CCII). You must look at the specifications page to get general information on the EIA connection. Also included are complete step by step installation instructions for the serial board.

I also purchased a TECHNICAL MANUAL, but have had very little use for it as no problems have developed in the printer. It does appear to be another excellent publication, with plenty of illustrations, a fairly good schematic, complete parts list, lubrication instructions and much more. The cost of this manual was suprisingly low, compared to what we are accustomed to paying for maintenance manuals of any type.

INTERFACE: Thanks, in part, to Ben Barlow's article "The Serial Port" (COLORCUE, AUG/SEP 1981), The interfacing was not a big problem. I had to add the Handshake Modification to the CCII and determine which pin # to use going into the printer for handshaking, wire up a db-25 connector and plug it in. I must admit that this is the one area I used the printer's Technical Manual as it has a good explanation of the serial interface.

PRINTER FEATURES: Font Styles include Standard, Italic and eight international character sets. Font Pitches are Pica, Elite and Condensed (136 columns per line), double-width (5, 6 and 8.5 CPI). SOME MORE FEATURES: Double-strike, Emphasized, Underline, Superscript, Subscript, Unidirectional, pre-set linefeed to almost any value, Form Feed, Variable form length (# of lines or Inches), Variable Header location, Vertical tab, Horizontal Tab, Back Space, Graphics - (Normal, Double, Quadruple-density), Macro instruction, Downloadable characters (make your own), and more.

Since I was told that it is EPSON compatible, I gambled several hours of programming to assemble Martin P. Rex's Screen Dump program (FORUM, SEP/OCT 1982) and try out the graphics. Mr. Rex's program and the Gemini will reproduce any and every character you can put on the CCII screen (in black & white, of course - unless you use some other color ribbon).

Speaking of ribbons, there is nothing special or expensive about the ribbon used in the Gemini. Even though discouraged by the salesman, I have been using up my stock of Teletype ribbons. I am careful to look for deposits on them before I put one on the printer. Technically speaking, it is a Standard Underwood spool-type, 13x50mm.

The Gemini 10X handles tractor feed paper (fanfold) 3-10 inches, Roll paper 8.5-10 inches (5 inch Dia.) and single sheets 8-10 inches.

The Gemini does all the Ads say it will and does it very well. It is of course, Dot Matrix but even that is hard to notice with a good ribbon. I am well satisfied with it's performance to date. The only question remaining is that of reliability. I have been through several ribbons and probably 10000 sheets of paper without a problem. The only failure I have heard of was almost immediate and the printer was replaced when it was returned.

Telling the Gemini what to do is as easy as typeing "plot 27,52" (print in italics) or "plot 27,69" (print in emphasized mode). Gemini also reconizes some control codes i.e. 14(A7 on) causes the printer to print in enlarged mode for one- line only, 18 (green)- pica, 19(yellow) - takes printer "off-line, 17(red)- puts printer back "on-line" and etc.

Since I have had no other quality printer I cannot compare the Gemini, but In case you haven't noticed, I am as pleased with the Gemini as I am with the CCII. All I need now is a Word Processor that will take advantage of all the Gemini features.

A Deluxe Keyboard Aid

Steve Perrigo
16925 Inglewood Road NE
B-306
Bothell, WA 98011

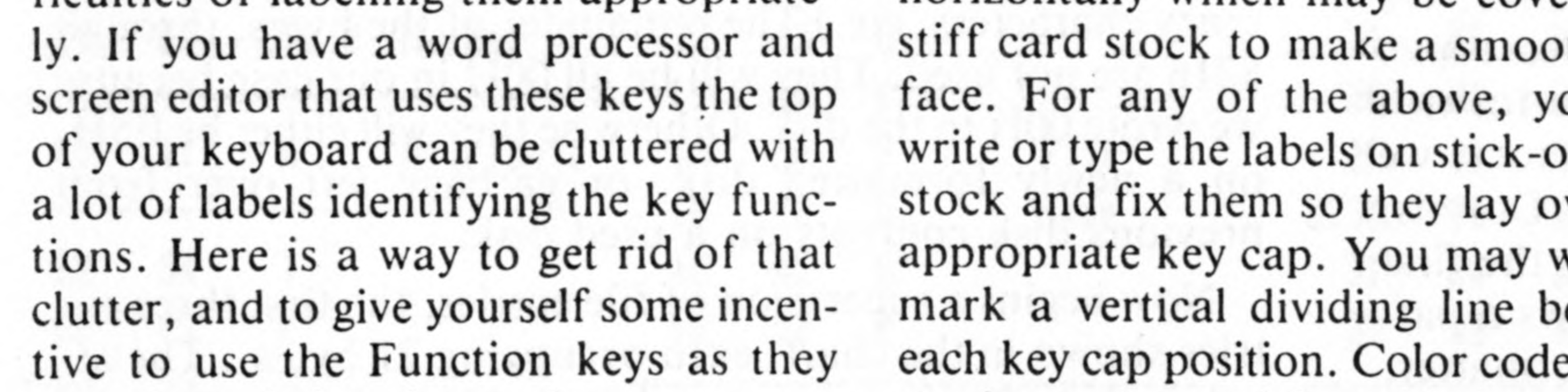
For those of you with the "deluxe" keyboard, the one with the 16 Function keys, this project is a 'must.' You have all probably wanted to use the Function keys for a program but haven't implemented them because of the difficulties of labelling them appropriately. If you have a word processor and screen editor that uses these keys the top of your keyboard can be cluttered with a lot of labels identifying the key functions. Here is a way to get rid of that clutter, and to give yourself some incentive to use the Function keys as they were intended to be used.

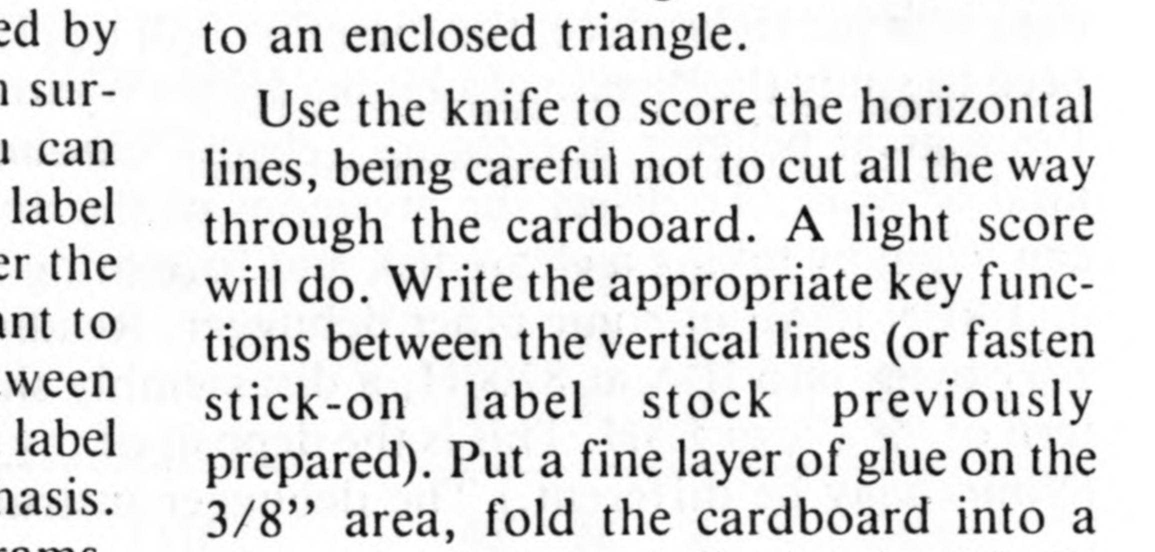
The idea is to fabricate a 'prism' that lies on the keyboard cover just above the row of Function keys, with a function description written on the prism just above each key. Each face of the prism holds the key codes for a different program, three programs for each prism. This method permits key labels large enough to read comfortably.

Such a prism may easily be made in several ways. Plastic supply houses often sell triangular solid stock. Have them cut a few 12-3/4" pieces for you. If that appears too expensive, you may want to search local drug stores for cheap plastic engineer's scales, triangular scales. These usually have finger grips cut horizontally which may be covered by stiff card stock to make a smooth surface. For any of the above, you can write or type the labels on stick-on label stock and fix them so they lay over the appropriate key cap. You may want to mark a vertical dividing line between each key cap position. Color coded label stock may be used for added emphasis. Some users color code their programs, white for the word processer side, green for the screen editor side, etc.

The prism may also be constructed from cardboard stock entirely. Using a sharp X-Acto type knife, cut the cardboard to 12-3/4" by 2-1/4". With a drawing pen, ink 16 vertical lines every 3/4" to make the divider between the seventeen keys on the Function key row (this includes the UP ARROW key.) Measuring from one long edge of the cardboard, ink three lines horizontally every 5/8" to mark the sides of the 'prism.' A 3/8" strip will be left at the other edge. This will become a glueing surface for fastening the cardboard into an enclosed triangle.

Use the knife to score the horizontal lines, being careful not to cut all the way through the cardboard. A light score will do. Write the appropriate key functions between the vertical lines (or fasten stick-on label stock previously prepared). Put a fine layer of glue on the 3/8" area, fold the cardboard into a triangular shape and slip the glued 3/8" surface under the open side of the triangle to close it. A narrow piece of wood and the table surface can be your clamps to hold the triangle closed until the glue sets. Instead of glue, you may be able to use double sided masking tape, if your cardboard is not too stiff. To seal the open ends of the 'prism' cut some triangular pieces from 3/16" balsa wood (available at any hobby store) and glue them in place. □





PeSTICidAl

"ERLOZGRMT GSV RMMVI HZMXGFN"

Our cliff-hanger in the last article was the prospect of forcing a directory entry for CYPHER.PRG, composed on IDA and written to the disk. Those of you who suspected this was pure braggadocio are in for a pleasant surprise. Let us first summarize where we were when we stopped.

Using an uninitialize disk, we used the WRIte command to write the code of CYPHER to the disk, beginning at block 0005. We then initialized the disk with five directory blocks. Next, through Basic, we created the file CYPHER.PRG, allowing enough space to include all the file code (this was done with a FILE "N" statement). When we tried to RUN CYPHER we received an error message because some critical parameters had not yet been entered into the directory record. Furthermore, we had an overlay table written to disk, beginning at block 000A, with no directory entry at all. (But you have created one as homework, by now.)

The directory doesn't much care who writes the information in it, whether it be operating system routines or you and me. It has a simple format, and as long as all the right data is in the right place, the directory will be effective. We need to study the directory a bit before we try to create one. I'm a great believer in creating "clean" baselines for this kind of work. To disect the mysteries of the directory, we can begin by taking a clean disk and formating it. Do this, and RUN IDAE or some other debugger. Reading the first 80H bytes into IDA at 8200H, a disassembly shows a pattern of "e's", or E5H. This is the deposit of my formatter. (Yours may be different.) The debugger instruction is:

```
IDA>XREA 00 8200-8280, or from FCS>REA 00 8200-8280
```

[I won't always cite procedures for the other instruments each time. Refer to the last issue or the instruction manuals for a reminder.]

In order to get a "cleaner" base line, let's write some 00H's to the first block of the formatted, uninitialized disk.

```
IDA>F 8200 8500 00
```

This will clear computer memory from 8200 to 8500.

```
IDA>XWRI 00 8200-8500, or from FCS>WRI 00 8200-8500
```

This will write the 00H's (NOPs) to the disk. Now initialize the newly-formatted disk:

```
IDA>XINI CD0:TESTDISK 05, or from FCS>INI CD0:TESTDISK 05
```

This will initialize the disk with five directory blocks.

Now we can copy CYPHER.PRG;02, from our very first work disk, to the newly initialized disk. If you have a single drive, then use the COPY.PRG software to make the transfer.

Activating IDAE again, we can now inspect the directory:

```
IDA>F 8200 8500 00; clear lots of computer memory to 00H.
IDA>XREA 00 8200-8280; read in the first directory block.
```

Within reasonable limits, you should see the contents of the directory as shown in Figure 1. The differences will be in the Free Space entry, because I am using an 8" double-sided disk which has considerably more space than the CD disk.

The first column of Fig. 1 shows the first 23 bytes of the directory. Most of them are not used for anything at all. The first byte is always 00H (NOP) for the first directory block. This byte states which directory block we are looking at, Block 0 being the first block, Block 1 being the second, and so on up to 04 for the fifth directory block. The second byte, labelled "Marker" in Fig 1, is always one less than the number of directory blocks specified in the INI command. Fig 1 shows I have five directory blocks on my disk (04 + 1). The third byte is always 41H, the ASCII letter "A", and is the attribute byte for the volume name. The next ten bytes, from 8203 to 820C contain the volume name. (You can put all kinds of things in there with IDA; colors, crazy characters, etc.) The remainder of the bytes, through 8216 are not used. They will be all 00H in our case because we wrote 00H to the disk. Otherwise they will either be E5H, on a newly formatted disk, or garbage left over from previous disk contents on a used disk.

Now begins a succession of file entries, the first three entries shown in the chart, each occupying 21 bytes.. The attribute byte for an unprotected file is 03H, always. Data involving two bytes, such as SBLK, SIZE, LADR, etc, are written low byte first. The contents, you will notice, follows the sequence that appears on the CRT directory listing. The "spare" byte, first appearing at 822B isn't used for anything, although numbers will sometimes be written there by the system software. I puzzled over this byte for a long time, failing to establish any correlation between its value and what was occuring with the disk file. I expected it to be a check sum for testing write integrity but I can't demonstrate that. (If you know something I don't know, please drop me a line.) A value of 00H is just fine for the "spare" byte, and has never failed to work for me with any kind of file.

Column three is the FREE SPACE entry for the directory.[1] Its attribute byte is 01H, always. There is no other meaningful data until SBLK, SIZE and LBC. LBC is always 80H for the FREE SPACE entry, so it's no problem determining that. But SBLK and SIZE should be accurate. Notice that SBLK is 000A, the first free disk byte following CYPHER, which confirms what we wrote to our uninitialized disk last time. (This was the beginning of the overlay for CYPHER's alternate encoding table.) The SIZE data in your directory will be a function of your drive type, CD drives having less free space than MD or FD drives.

ProGRammIng!!!

Part Two

W. S. Whilly
(Wherever)

The last valid directory entry is always the free space entry, and it counts as one of the files in the directory. My directory blocks hold five files, which means I can store four "real" files and one FREE SPACE "file." You can always identify the FREE SPACE entry by the presence of the attribute byte (01H).

Column four is unused, and contains all 00H. Should I want to add another "real" file to the directory, I would add it beginning at the byte at 822CH, and move my FREE SPACE entry, now amended as to attribute, SBLK and SIZE, to the byte beginning at 8241H in Column 4 of Fig 1.

So let's go to work, and fix our last disk from the previous article. The directory should currently look like this (unless you really did create a file entry for the overlay):

FIG 1. MAP OF DISK DIRECTORY BLOCK

-----DISK VOLUME SPACE-----

ADDR	BYTE	TRANS	FUNCTION
8200	0	@	NOP
01	04	CTL D	Marker
02	41	A	Attribute
03	54	T	
04	45	E	
05	53	S	
06	54	T	Volume
07	44	D	Name
08	49	I	
09	53	S	
0A	4B	K	
0B	20	space	
0C	20	space	
0D	00	@	
0E	00	@	
0F	00	@	
8210	00	@	Not
11	00	@	Used
12	00	@	
13	00	@	
14	00	@	
15	00	@	
16	00	@	

-------1ST FILE ENTRY-------

ADDR	BYTE	TRANS	FUNCTION
8217	03	CTL C	Attribute
18	43	C	
19	59	Y	
1A	50	P	File
1B	48	H	Name
1C	45	E	
1D	52	R	
1E	50	P	File
1F	52	R	Type
8220	47	G	
21	01	1	Version
22	05		SBLK
23	00		0005
24	05		SIZE
25	00		0005
26	80		LBC
27	00		LADR
28	82		8200
29	00		SADR
2A	82		8200
2B	00		Spare

-------2ND FILE ENTRY-------

ADDR	BYTE	TRANS	FUNCTION
822C	01	CTL A	Attribute
2D	00	@	(Free Space
2E	00	@	Entry)
2F	00	@	File
8230	00	@	Name
31	00	@	
32	00	@	
33	00	@	File
34	00	@	Type
35	00	@	
36	00	@	Version
37	0A		SBLK
38	00		000A
39	02		SIZE
3A	12		1202
3B	80		LBC
3C	00		LADR
3D	00		
3E	00		SADR
3F	00		
8240	00		Spare

-------3RD FILE ENTRY-------

ADDR	BYTE	TRANS	FUNCTION
8241	00	@	Attribute
42	00	@	
43	00	@	
44	00	@	File
45	00	@	Name
46	00	@	
47	00	@	
48	00	@	File
49	00	@	Type
4A	00	@	
4B	00	@	Version
4C	00	@	SBLK
4D	00	@	
4E	00	@	SIZE
4F	00	@	
8250	00	@	LBC
51	00	@	LADR
52	00	@	
53	00	@	SADR
54	00	@	
55	00	@	Spare

```
DIRECTORY DF0: TESTDISK 05

03 CYPHER.PRG;01 0005 0005 80 0001 0280
01 FREE SPACE>   000A 1202
```

Let's reinitialize this disk (no, it won't hurt the files we have on it!) and start from "scratch," making the directory conform to our needs. Go ahead! Have faith!

```
IDA>XINI CD0:TESTDISK 05
```

Now print the new directory on the screen and write down the SIZE of the FREE SPACE entry for future use:

```
IDA>XDIR ; (my FREE SPACE is 1207H blocks.)
```

Let's clear some computer memory,

```
IDA>F 8200 8500 00
```

and read in the first block of the new directory:

```
IDA>XREA 00 8200-8280
```

You may now do a disassembly or hex dump from 8200 to 8280, and fill in the values you find there in the first column of the chart in Fig 2 (from addresses 8200 to 8216 only).

Using the chart of Fig 2 as a worksheet, and working lightly in pencil at first, let's prepare to construct a directory for our disk, making entries for CYPHER.PRG and the overlay. Working now in column 2, the first directory entry, we enter the value 03H at address 8217, because the attribute for a "real" file is always 03H.

From 8218 we enter the file name, CYPHER, in hex, as 43H, 59H, 50H, 48H, 55H, 52H. Beginning at address 821E, we enter the file type, PRG, as 50H, 52H, 47H. At address 8221 we enter 01H for the version. (Are you doing this as we go?!)

At address 8222, we enter the SBLK (start block), low byte first, 05H, 00H (=0005H), and the SIZE of CYPHER (5 blocks, remember?) at address 8224; 05H, 00H (= 0005H).

We settled on a (L)ast (B)lock (C)ount of 80 for CYPHER, last time, even though that's not entirely correct; but we can use it. At address 8226, enter 80H. Now the fun begins. At address 8227, we enter the desired LADR (loading address) and it will be 8200, but low byte first: 00H, 82H. We can get fancy with the starting address. Remember that CYPHER begins with three NOPs that don't do anything. At address 8229, enter "8203", low byte first: 03H, 82H. Put a 00H in the "spare" byte slot (=00H) and we've done it!

But wait! We must still make a FREE SPACE entry. So move over to the next file column, and put the correct attribute byte at address 822C. What did you put there? A 01H, of course. 01H is the attribute byte for FREE SPACE. Now add the SBLK at address 8237, low byte first. 05H (for the directory) + 05H (for CYPHER) = 0A as the next block. (5 + 5 is A, in hex.)

The SIZE will be the free space you noted from the freshly initialized directory (1207 for me) minus the blocks we just assigned to CYPHER (=05H). So the FREE SPACE SIZE is, for me, 1207-0005 = 1202. I will put 02H, and 12H into my chart beginning at address 8239. Put 80H in the LBC slot at address 823B.

Let's put the values from the chart into memory. With IDA it's very easy:

```
IDA>P 8200
```

Move the cursor right or left with the arrow keys. As you enter hex numbers, the cursor will automatically move to the next slot on the right. When you reach the end of the line, the next set of addresses are automatically displayed. Begin at 8200 and keep on entering data, checking addresses and contents as you go, until you reach 823B, the LBC data for FREE SPACE.

Check it out with a hex dump from 8200 to 8240.

If it all checks out, write the directory you just made to disk:

```
IDA>XWRI 00 8200-8280
```

Display the directory:

```
IDA>XDIR
```

Run CYPHER from your new directory:

```
IDA>XRUN CYPHER
```

WOW! Now return to IDA and clear 8200-8500 with 00H again. Read the directory back in:

```
IDA>XREA 00 8200-8280
```

Using the Chart of Fig 2, erase the numbers in column three, the FREE SPACE column. We will now create a directory entry there for the overlay.

At address 822C, we must change the attribute from 01H to 03H. We can now enter the file name, TABLE1.OVR;01 in succeeding bytes. (That's this succession of hex numbers, my friends: 54, 41, 42, 4C, 45, 31, 4F, 56, 52, 01.)

SBLK is the same is it was, 000AH, low byte first. Enter at 8237H. Enter the SIZE as one block, 0001H. The LBC must be calculated somehow. LBC tells how many of the 128 disk block bytes are being used. It does not mean how many are left over! An LBC of 80H means all 128 bytes in the block are used. You will have 34H bytes or so in your overlay, depending on whether or not the last two space bytes were saved with the table. Put LBC at 823B.

The loading address is 8448H. That's where the table begins, and that's the point we saved it from (see last Colorcue). But what about the start address?

SADR is used by the system ROM to get a LDA or PRG program by placing this number in the program counter. ROM will not be looking for SADR in our file type OVR, which we just made up. We can make SADR 000H, then. We prevent SADR from causing trouble by LOADing the OVR, which simply puts the bytes in memory, instead of RUNning it, which would cause ROM to look for a starting address.

We need to put 00H into the "spare byte" slot at address 8240. Now it's your time to make the necessary FREE SPACE entry in column four, the third file entry. After you've done it, refer to [3] for my answers.

Write the new directory to disk, right over the old one:

```
IDA>XWRI 00 8200-8280
IDA>XDIR
IDA>LOAD CYPHER>PRG;01
IDA>LOAD TABLE1.OVR
IDA>G 8200
```

There it is, folks! You have broken into the inner sanctum!

But what happens when we look beyond the fifth directory entry? We will find two bytes preceeding the sixth entry. The first of these will be 01H, indicating that we are in the second directory block (of five.) The second byte will be 04H, the total number of directory blocks minus one. The following byte will be the attribute byte for the sixth directory entry.

It should be clear from this exercise that you now have a ready tool for reconstructing a clobbered disk directory....IF... you have a printout of the directory to work from. I periodically make a directory printout of all my important disks and store it in the disk sleeve. Not only is this a convenient way to view the disk contents, but it is a good way to have the data on hand for a reconstruction, ... and who hasn't needed a reconstruction at one time or another.

Even without a directory printout, IDA can search a clobbered disk, one block at a time, locate programs and create

FIG 2. DIRECTORY WORKSHEET

-----DISK VOLUME SPACE-----				-------1ST FILE ENTRY-------				-------2ND FILE ENTRY-------				-------3RD FILE ENTRY-------			
ADDR	BYTE	TRANS	FUNCTION	ADDR	BYTE	TRANS	FUNCTION	ADDR	BYTE	TRANS	FUNCTION	ADDR	BYTE	TRANS	FUNCTION
8200	0	a	NOP	8217			Attribute	822C			Attribute	8241			Attribute
01	04	CTL D	Marker	18				2D				42			
				19				2E				43			
02	41	A	Attribute	1A			File	2F			File	44			File
				1B			Name	8230			Name	45			Name
03	54	T		1C				31				46			
04	45	E		1D				32				47			
05	53	S													
06	54	T	Volume	1E			File	33			File	48			File
07	44	D	Name	1F			Type	34			Type	49			Type
08	49	I		8220				35				4A			
09	53	S													
0A	4B	K		21			Version	36			Version	4B			Version
0B	20	space													
0C	20	space		22			SBLK	37			SBLK	4C			SBLK
				23			0005	38			000A	4D			
0D	00	a													
0E	00	a		24			SIZE	39			SIZE	4E			SIZE
0F	00	a		25			0005	3A			1202	4F			
8210	00	a	Not												
11	00	a	Used	26			LBC	3B			LBC	8250			LBC
12	00	a													
13	00	a		27			LADR	3C			LADR	51			LADR
14	00	a		28			8200	3D				52			
15	00	a													
16	00	a		29			SADR	3E			SADR	53			SADR
				2A			8200	3F				54			
				2B			Spare	8240			Spare	55			Spare

new directory listings for them. If you know anything at all about your programs, the construction of their source code, their text, or whatever, IDA will assist in retrieving them from a destroyed disk. It would be helpful to make a "dry run" with CYPHER.PRG, by reinitiallizing the disk we have just created, and by means of a block to block search, get CYPHER back into computer memory and SAVE it.

A review of FCS SAVE command will also be useful.[2] It is often used to save memory contents as a PRG file. You will have noticed that when we use the FCS REAd or WRIte commands they take a similar format: for example:

```
IDA>XREA 00 8200-847D; Note dash between last two numbers.
```

This means "read beginning at block 00 into memory starting at address 8200, and continue reading until memory is filled to address 847D." The dash between 8200 and 847D indicates that both numbers are memory addresses. If the dash is omitted, the last number, 847D, indicates how many bytes are to be read—in this case far too many for the purpose. FCS allows you to specify either the last memory address to be filled (by using the dash) or the number of bytes to be read (without the dash.) This same convention applies to the SAVE command as well. The SAVE command also permits you to specify LADR and SADR, and this is useful for converting LDA files to PRG from IDA. The format is a little tricky. Here are some possibilities for a mythical LDA file, CYPHER.LDA:

```
Example 1 - SAVE CYPHER.PRG;01 8200-847D 8203
```

This tells FCS to save the code beginning at 8200 and ending at 847D to a disk file. LADR will be 8200 and SADR will be 8203.

```
Example 2 - SAVE CYPHER.PRG 8200-847D
```

We omitted the version number this time, so FCS will supply one for us. Since we omitted a specific SADR, FCS will make SADR the same as LADR, in this case 8200.

```
Example 3 - SAVE CYPHER.PRG 8200 0280 8203 9000
```

This is a most sophisticated instruction. We have changed the memory specification to show, not the end address in memory (847D in the first examples), but the number of bytes (0280H) to be saved. We have indicated SADR is to be 8203H. But the last number is telling FCS that the code we want to save isn't currently located in memory at 8200 at all. It is really in memory beginning at 9000, but we want it to read from disk to memory, henceforth, with LADR = 8200. FCS will write 0280H bytes, beginning at 9000 onto the disk, label it CYPHER.PRG and set LADR = 8200, and SADR = 8203. What use is this? Not much, in fact, and a "neater" way would be to use IDA to relocate the code where we actually wanted it to be in memory before a SAVE to disk as a PRG file.

Use Example 2 if LADR and SADR are to be the same. Use Example 1 if you want to specify an SADR not the same as LADR.

We have not exhausted the potential of IDA by a long shot, and we'll continue next time with the monitor discussion I promised for this time. (The editor won't give me any more room.) But one of IDA's most useful features is the reports it can generate to the printer. Any screen display, from the top of the screen to the current cursor position may be dumped to the printer by simply pressing CMD/PRINT. If you do not have the extended keyboard, this may be simulated by holding down at the same time the following three keys: SHIFT/CONTROL/V (cyan color key).

To set the port Baud rate, type from the IDA prompt some form of Bn(2), where n = 1 to 7, and the optional (2) adds two stop bits, example;

```
IDA>B7 ; for a Baud of 9600 and 1 stop bit.
```

IDA will also permit you to write on the CRT, in a simulated CRT mode, to make notes on the screen before you dump it! RUN IDAE, and XLOAD CYPHER.PRG. Disassemble from 8200 15+ to get a screen display. Now enter the simulated CRT mode by pressing the BREAK key, followed by CMD/CRT (COMMAND key and SHIFT/CRT all at the same time.) You may now use the cursor control keys to position the cursor anywhere on the screen, type your messages, then press ESC to return to the IDA prompt. CMD/PRINT will now dump the edited screen to the printer. Several of the printouts in my first article were constructed in this way. If you haven't ordered IDA yet, there's still time. Much more fun to come! W. S. Whilly. □

[1] If there were formerly a "real" file in this entry column on your disk, the file name and type may still be visable. The DELete command does not erase these parameters, but the attribute byte will be 01H, telling FCS that this is, indeed, the FREE SPACE entry.

[2] This information has been published previously by Jim Minor in *DATA CHIP*, -29, Dec/Jan 1982. Jim has a thorough presentation here of the REA, WRI, SAVE and LOAD commands that has not been published elsewhere. I highly recommend this article as a clear and thoughtful presentation of this material.

[3] SBLK = 000B; SIZE = 1202-1 = 1201; LBC = 80.[172]

FIG 3. Hex Dump of Directory with CYPHER.PRG as the only file. Note FREE SPACE entry.

```
IDA>H 8200 8280

8200  00 04 41 54 45 53 54 44  49 53 4B 20 20 00 00 00
8210  00 00 00 00 00 00 00 03  43 59 50 48 45 52 50 52
8220  47 01 05 00 05 00 80 01  00 80 02 01 01 00 00 00
8230  00 00 00 00 00 00 00 0A  00 02 12 80 00 00 00 00
8240  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
8250  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
8260  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
8270  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
8280  00
```

ITEM	PRICE
COMPUTERS:	
Intelligent Systems Corporation:	
Model 3651, 32K RAM, 117 key keyboard, lower case	ask for price
Morrow Micro Decision:	
MD2, CP/M computer system, 2 diskdrives, single sided, 190 K, with Wordstar wordprocessor	$ 1,490
MD3, CP/M computer system with 2 diskdrives each 386K, Liberty monitor, with complete set of business software	$ 1,845
MD11, complete 11 mbytes hard disk computer system, high resolution graphics monitor, complete set of business software, ...	$ 2,645
NEC APC	
True 16bit CP/M86 (or MS.DOS) computer system, with high resolution graphics monitor, 2 8" diskdrives each 1 mbyte, ...	$ 2,795
USED CCII 32K incl. software	$ 500
USED KAYPRO II incl. software	$ 1,300
PERIPHERALS AND OPTIONS:	
Bell kit and simple soundware kit for 3651	$ 25
CCII RS232 CTS kit "handshake"	2
Lower case character kit, switchable	38
Joysticks with instruction manual	33
Bank board 56 K EPROM, software selectable	286
Disk drive 5 1/4" for 3651 cable included	350
Disk drive 5 1/4" for CCII V6.78 cable incl.	250
Disk drive 5 1/4" for CCII V8.79 cable incl.	250
Keyboard upgrade kit for CCII, 72 keys to 117 keys	150
Keyborad upgrade kit for 3651, 72 keys to 117 keys	250
Wordprocessor keycaps	31
PRINTERS:	
Gemini 10X dot matrix printer	359
Gemini 15X dot matrix printer	495
RS232 Serial interface board	55
Brother HR-15 daisy wheel printer with sheetfeeder	852
Cable for printer to computer	25

**** all items subject to availability ****

VISA, MASTER CHARGE AND AMERICAN EXPRESS ACCEPTED

ICS ************************ INTELLIGENT COMPUTER SYSTEMS INC.

ICS INTELLIGENT COMPUTER SYSTEMS INC.

12117 COMANCHE TRAIL • HUNTSVILLE AL 35803 • PHONE 205 881-3800

Disk Salvage

Bob Mendelson
27 Somerset Place
Murray Hill, NJ 07974

[Because of differing ROM calls and memory mapping, this program is not suitable for the CCII. Refer to W.S. Whilly's article, this issue, for a suitable equivalent procedure for the CCII. ed.]

The unexpected happened. I intended to initialize a new disk in drive #1 but I forgot to type in the '1', and lo and behold I had reinitialized a utility disk with 30 programs. 'DIR' only printed out an empty disk. I knew that INI does not wipe out the disk in the same way that formatting does, but I had only a vague idea of how the Directory is constructed. To explore this, the first 9 sectors of the directory were loaded into memory at A000H by use of the REA command. From here on it was easy.

The first 16 bytes are used for the ID of Sector 0, the name of the disk, followed by 10 'don't care' bytes. It was also apparent that only the first 80H bytes of Sector 0 are cleared to 00H. Everything else was unchanged; that is, the rest of the directory entries, the final line that shows the sectors used, the number left, and the delimiter, 80H (LBC), and all the program code.

My first try was to type in the data for the first 5 programs by use of the DIR printout that had been made for my library reference. This was done with the CPU monitor and an ASCII table for letters. It was primitive but not difficult. Each line of the directory uses 15H bytes, the last one of which is a 'don't care' spare byte. Therefore, new lines start at addresses A017, A02C, A041, A056, A06B... ending at A07F. A080 has a two-byte ID for Sector 1, which is followed by another set of 15H data blocks. Following the last line of active directory entries, there is an 01H, followed by 10 bytes, each 00H, and then the calculated number of sectors used, the number remaining, and an 80H delimiter.

The program in Listing 1. was written to allow simple reentry of the first five lines of the directory. Following reentry, the program will then write the data onto the disk and call for a directory printout. Should the directory have four or less lines, the last line will have been wiped out. Therefore, after typing the last data line, an 01H at the start of the following line will automatically calculate the used and free sectors and write them to the disk.

To save publishing space for the Listing, I have omitted the program instructions from the SRC code. They are printed here instead:

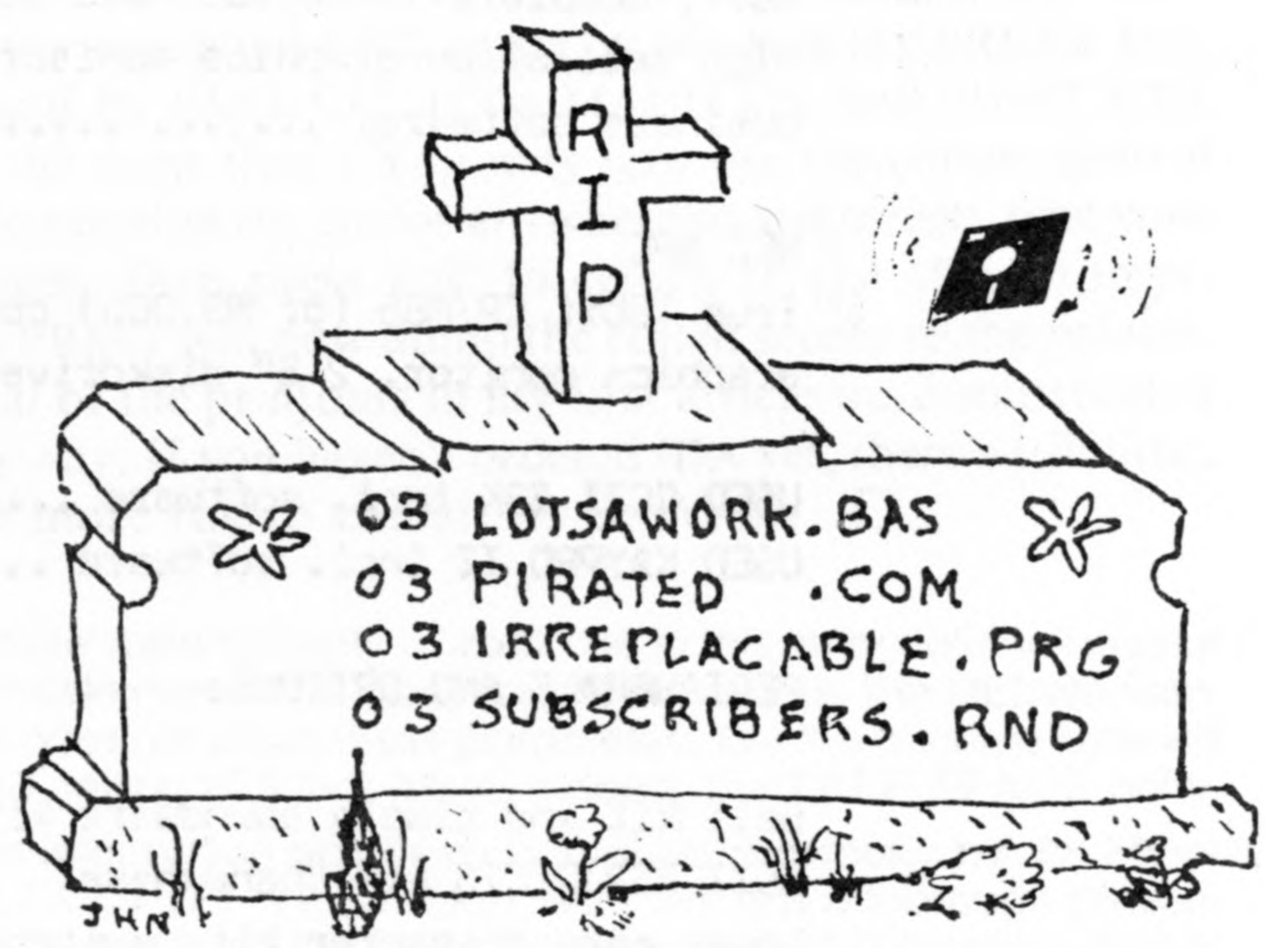
INSTRUCTIONS FOR UNLOCK.PRG -

THIS PROGRAM WILL RECOVER A DISK THAT WAS INITIALIZED BY MISTAKE IF A PREVIOUS PRINTOUT OF THE DISK DIRECTORY IS AVAILABLE CONTAINING THE ORIGINAL DIRECTORY INFORMATION.

TO OPERATE THIS PROGRAM, ENTER THE COMPLETE DATA FROM EACH LINE AS IT APPEARS ON THE PRINTOUT. DO NOT PRESS 'RETURN' UNTIL THE ENTIRE LINE IS ENTERED. IF NO PRINTOUT IS AVAILABLE, USE THE 'REA' COMMAND AND ESC P TO DETERMINE THE START OF EACH PROGRAM BLOCK AND THE PROGRAM SIZE.

THE LAST BLOCK COUNT (LBC) MAY BE 80H IF IN DOUBT. IF LADR (LOAD ADDRESS) AND SADR (START ADDRESS) ARE NOT KNOWN, CHOOSE ONE AND LATER CHANGE TO THE CORRECT ADDRESS. ONLY ALPHABETICAL CHARACTERS MAY BE EDITED DURING DATA ENTRY UNDER CURSOR CONTROL. DO NOT ATTEMPT TO CHANGE BEYOND THE ';' FOLLOWING THE FILE TYPE. NUMBERS ARE IN HEX, AND HANDLED IN THE SAME WAY THAT THE MONITOR HANDLES THEM; THAT IS, ONLY THE LAST FOUR HEX DIGITS WILL BE ACCEPTED BY THE PROGRAM. ENTER A SPACE TO SEPARATE NUMERICAL ENTRIES ON EACH LINE.

IF THE TOTAL NUMBER OF LINES IS FOUR OR LESS, TYPE '01' FOR 'ATR' TO END THE INPUT AND HAVE THE PROGRAM AUTOMATICALLY FILL IN THE 'FREE SPACE' DATA. IF FIVE LINES ARE TYPED IN, THE 'FREE SPACE' DATA WILL BE SUPPLIED WITHOUT MANUAL HELP. □



LISTING 1 Disk Salvage: UNLOCK
By R. Mendelson, V6-84

```
;A program to recover contents of a disk that was
; initialized in error. Only the first 80H bytes
; need to be restored to recover the directory.
; Remaining directory blocks & programs are intact.

;Equates.......................................

A017  BUFF    EQU   0A017H  ;Store DIR string
0103  CI      EQU   0103H   ;Console in
0109  CO      EQU   0109H   ;Console out
0100  CPUOS   EQU   0100H   ;CPU Monitor
000D  CR      EQU   13      ;Carriage return
0E74  CRLF    EQU   0E74H   ;Next line + GRN FG
00EF  EOS     EQU   239     ;'End of string'
0133  EXPR    EQU   0133H   ;Convert ASCII -> HEX

0012  GR      EQU   18      ;GREEN
000A  LF      EQU   10      ;Line feed
010F  LO      EQU   010FH   ;Character to CRT
9FFF  KEYBF1  EQU   9FFFH   ;KEYBOARD READY flag
012A  OSTR    EQU   012AH   ;Print string
```

```
0011  RED      EQU     17         ;RED
0020  SPACE    EQU     20         ;SPACE
0013  YEL      EQU     19         ;YELLOW
0F27  VECT     EQU     0F27H      ;Restart vector

;.............................................

;Note: Leading zeros may be omitted from hex inputs.

0000               ORG     0B000H

B000 AF        START:  XRA     A
B001 32FF9F            STA     KEYBF1
B004 CD0301            CALL    CI
B007 2149B1            LXI     H,MSG1A ;Disk warning
B00A CD2A01            CALL    OSTR    ;Print it.

;Note: If 01 is hit before adding any line to the
; directory, Free Space SBLK will be set at 0009
; & SIZE at 0000.

B00D 3E09      PROTEK: MVI     A,9
B00F 3235B2            STA     SBLKX   ;Set SBLK=0009
B012 AF                XRA     A       ;Clear Acc
B013 3236B2            STA     SBLKX+1 ;Set SIZE to
B016 3237B2            STA     SBLK+2  ; 0000H and
B019 3238B2            STA     SBLK+3  ; store it.

B01C 2117A0    START1: LXI     H,BUFF  ;Load ptr
addr
B01F 2232B2            SHLD    ADDR1   ;Save it
B022 32FF9F            STA     KEYBF1  ;(A=0)
B025 CD0301            CALL    CI

B028 21C9B1            LXI     H,MSG2  ;Column headings
B02B CD2A01            CALL    OSTR    ;Print them

B02E 2117A0            LXI     H,BUFF
B031 AF                XRA     A
B032 0668              MVI     B,7FH-17H

;Set counter just past directory name and fill unused
; bytes with 00H....

B034 77        FILL:   MOV     M,A     ;Byte to memory
B035 23                INX     H       ;Index pointer
B036 05                DCR     B       ;Decr counter
B037 C234B0            JNZ     FILL    ;Fill next memory

B03A 21F9B1    READ1:  LXI     H,MSG3  ;Point to MSG
B03D CD2A01            CALL    OSTR    ;Print it
B040 3E05              MVI     A,5     ;Number of lines
B042 3234B2            STA     COUNT   ; to be typed in.

B045 3E20      SCREEN: MVI     A,SPACE ;Set CRT position
B047 CD0F01            CALL    LO
B04A 3E30              MVI     A,30H   ;Fake zero
B04C CD0F01            CALL    LO

B04F CDB1B0    ATR:    CALL    Y1      ;Input ATR
B052 FE31              CPI     31H     ;Is it 01H?
B054 CADEB0            JZ      FREE    ;Yes, set Free
Space
B057 D630              SUI     30H     ;No, ASCII to HEX
B059 2B                DCX     H       ;Back to byte #1
B05A 77                MOV     M,A     ;Insert hex #
B05B 23                INX     H       ;..for next char
B05C 3E20              MVI     A,SPACE ;Insert CRT space
B05E CD0F01            CALL    LO      ;Print it.

B061 CDBAB0    NAME:   CALL    Y2      ;Input file name
B064 3E07              MVI     A,7     ;Check if all
B066 B8                CMP     B       ; chars are in.
B067 C261B0            JNZ     NAME    ;No, go back.
B06A 3E2E              MVI     A,'.'   ;Add TYP delimit
B06C CD0F01            CALL    LO

B06F CDBAB0    TYPE:   CALL    Y2      ;Get file TYP
B072 3E0A              MVI     A,10    ;3 more bytes in?
B074 B8                CMP     B
B075 C26FB0            JNZ     TYPE    ;No, go back
B078 3E3B              MVI     A,';'   ;Add TYP delimit
B07A CD0F01            CALL    LO

B07D CD2EB1    VERS:   CALL    HEXNU1  ;Get version in.
B080 2B                DCX     H       ;Overlay MSB w/LSB

B081 CD2EB1    SBLK:   CALL    HEXNU1  ;Get SBLK
B084 EB                XCHG            ;From DE to HL
B085 2235B2            SHLD    SBLKX   ;Save it.
B088 EB                XCHG            ;Back to DE

B089 CD2EB1    SIZE:   CALL    HEXNU1  ;Get file soze
B08C EB                XCHG
B08D 2237B2            SHLD    SIZEX
B090 EB                XCHG

B091 CD2EB1    LBC:    CALL    HEXNU1  ;Get LBC
B094 2B                DCX     H       ;Overlay MSB of LBC

B095 CD2EB1    LADR:   CALL    HEXNU1  ;Get loading addr

B098 CD2EB1    SADR:   CALL    HEXNU1  ;Get SADR from
Keybd
B09B 23                INX     H       ;Pass spare byte
B09C 2232B2            SHLD    ADDR1   ;Save ptr addr
B09F 3A34B2            LDA     COUNT   ;Get prev line cnt
B0A2 3D                DCR     A
B0A3 FE00              CPI     0       ;See note below
B0A5 CA1FB1            JZ      FREE2
```

```
;Note: If 5 lines are being typed, stop before adding a
; 5th line 'Free Space' line, since the rest of the
; directory is still intact.

B0A8 3234B2          STA     COUNT
B0AB CD740E          CALL    CRLF    ;For next dir line.
B0AE C345B0          JMP     SCREEN  ;Start next line.

;Input handler...................................

B0B1 2A32B2    Y1:   LHLD    ADDR1   ;Current buff addr
B0B4 0600            MVI     B,0     ;Reset char counter
B0B6 AF              XRA     A       ;A=0
B0B7 32FF9F          STA     KEYBF1  ;Clear keybd flag
B0BA CD0301    Y2:   CALL    CI      ;Input character
B0BD FE1A            CPI     1AH     ;Back space?
B0BF CAC9B0          JZ      Y4      ;Yes, jump

;Input to buffer.................................

B0C2 77              MOV     M,A     ;Char into buffer
B0C3 23        Y3:   INX     H       ;Incr buffer addr
B0C4 2232B2          SHLD    ADDR1   ;Update buff ptr
B0C7 04              INR     B       ;Incr counter
B0C8 C9              RET             ;..for next char.

;Backspace routine...............................

B0C9 2B        Y4:   DCX     H       ;Back up buff ptr
B0CA 05              DCR     B       ; & start of line.
B0CB C2C9B0          JNZ     Y4      ;Do again!
B0CE 2232B2          SHLD    ADDR1   ;Update buff ptr
B0D1 3E0B            MVI     A,0BH   ;Erase line.
B0D3 CD0F01          CALL    LO
B0D6 3E0D            MVI     A,CR    ;Carriage return
B0D8 CD0F01          CALL    LO
B0DB C345B0          JMP     SCREEN  ;Start line over.


;Insert FREE SPACE entry if dir < 5 lines

B0DE D630      FREE:  SUI    30H     ;ASCII to HEX
B0E0 2B               DCX    H       ;Back 1 buff addr
B0E1 77               MOV    M,A     ;Number into buff
B0E2 23               INX    H
B0E3 AF               XRA    A       ;A=0
B0E4 060A             MVI    B,10    ;Set 10 blanks

B0E6 77        FREE1: MOV    M,A     ;Insert zero
B0E7 23               INX    H
B0E8 05               DCR    B       ;Reduce counter
B0E9 C2E6B0           JNZ    FREE1   ;If B<>0

;Calculate next free block (SADR).....

B0EC 2232B2           SHLD   ADDR1   ;Save buffer addr
B0EF 2A35B2           LHLD   SBLKX   ;Get last SBLK
B0F2 EB               XCHG           ;Save it in DE
B0F3 2A37B2           LHLD   SIZEX   ;Get SIZE
B0F6 19               DAD    D       ;Add for SBLK
B0F7 EB               XCHG           ; and move to DE
B0F8 2A32B2           LHLD   ADDR1   ;Current buffer
addr
B0FB 73               MOV    M,E     ;Place LSB in
buffer
B0FC 23               INX    H
B0FD 72               MOV    M,D     ;Place MSB in
buffer
B0FE 23               INX    H

;Calculate blocks still free (SIZE)...
; DE still has SBLK -

B0FF 2232B2           SHLD   ADDR1   ;Save buffer addr
B102 7B               MOV    A,E     ;Make 1's
complement
B103 2F               CMA            ; of E register
B104 5F               MOV    E,A     ;Save it.
B105 7A               MOV    A,D     ;1's complement of
B106 2F               CMA            ; D register
B107 57               MOV    D,A     ; and save it too.
B108 EB               XCHG           ;Complement into HL
B109 1E01             MVI    E,1     ;Set DE=0001
B10B 1600             MVI    D,0
B10D 19               DAD    D       ;Convert to 2's comp
B10E EB               XCHG           ;Move it to DE
B10F 2602             MVI    H,02H   ;Total sector count
B111 2E76             MVI    L,76H   ; is 276H
B113 19               DAD    D       ;Difference in HL
B114 EB               XCHG           ;Move it to DE
B115 2A32B2           LHLD   ADDR1   ;Current buffer
addr
B118 73               MOV    M,E     ;Insert LSB
B119 23               INX    H
B11A 72               MOV    M,D     ; and MSB.
B11B 23               INX    H


;Terminate directory .................

B11C 3E80             MVI    A,80H   ;Delimiter (LSB)
B11E 77               MOV    M,A     ; into buffer

B11F 210FB2    FREE2: LXI    H,MSG4  ;Put dir on disk
B122 CD2A01           CALL   OSTR
B125 2125B2           LXI    H,MSG5  ;Print to CRT
B128 CD2A01           CALL   OSTR
B12B C3270F           JMP    VECT    ;Return to system
```

```
;VECT is restart vector RST1, and the technical end of
; the program.

;Subroutines ......................................

B12E 2232B2   HEXNU1: SHLD   ADDR1      ;Save buffer
addr
B131 CD42B1           CALL   HEXIN
B134 EB               XCHG              ;Move HL to DE
B135 2A32B2           LHLD   ADDR1
B138 73               MOV    M,E        ;Insert LSB
B139 23               INX    H
B13A 72               MOV    M,D        ; and MSB
B13B 23               INX    H

;Note: DE retains the latest value for
; use in SBLK & SIZE -

B13C 3E20     HEXNU2: MVI    A,20H      ;'space'
B13E CD0F01           CALL   LO         ;CRT display only
B141 C9               RET

B142 0E01     HEXIN:  MVI    C,1        ;4-byte ASCII to 2.
B144 CD3301           CALL   EXPR       ;Hex # in HL
B147 E1               POP    H
B148 C9               RET

;String storage..................................

B149 0C135052 MSG1A:  DB     12,YEL,'PROGRAM TO REPAIR'
B14D 4F475241
B151 4D20544F
B155 20524550
B159 414952
B15C 204C4F53         DB     ' LOST DIRECTORY-',GR
B160 54204449
B164 52454354
B168 4F52592D
B16C 12
B16D 03000242         DB     3,0,2,'BY R. MENDELSON',YEL
B171 5920522E
B175 204D454E
B179 44454C53
B17D 4F4E13
B180 03000450         DB     3,0,4,'PLACE ',RED,'DISK TO '
B184 4C414345
B188 20114449
B18C 534B2054
B190 4F20
B192 42452052         DB     'BE REPAIRED ',YEL,'IN DRIVE-'
B196 45504149
B19A 52454420
B19E 13494E20
B1A2 44524956
B1A6 452D
B1A8 03200412         DB     3,32,4,GR,'(HIT ANY KEY TO '
B1AC 28484954
B1B0 20414E59
B1B4 204B4559
B1B8 20544F20
B1BC 434F4E54         DB     'CONTINUE)',CR,LF,LF,EOS
B1C0 494E5545
B1C4 290D0A0A
B1C8 EF

B1C9 41545220 MSG2:   DB     'ATR NAME TYPE VR  SBLK  SIZE'
B1CD 4E414D45
B1D1 20545950
B1D5 45205652
B1D9 20205342
B1DD 4C4B2020
B1E1 53495A45
B1E5 20204C42         DB     '  LBC LADR  SADR',CR,LF,LF
B1E9 43204C41
B1F1 53414452
B1F5 0D0A0A
B1F8 EF               DB     EOS

B1F9 1B045245 MSG3:   DB     27,4,'REA0:0 A000-A47F',13
B1FD 41303A30
B201 20413030
B205 302D4134
B209 37460D
B20C 1B1BEF           DB     27,27,EOS

B20F 1B045752 MSG4:   DB     27,4,'WRI0:0 A000-A47F',13
B213 49303A30
B217 20413030
B21B 302D4134
B21F 37460D
B222 1B1BEF           DB     27,27,EOS

B225 0D0A121B MSG5:   DB     CR,LF,GR,27,4,'DIR0:',27,27
B229 04444952
B22D 303A1B1B
B231 EF               DB     EOS
;Data storage....................................

B232          ADDR1:  DS     2
B234          COUNT:  DS     1
B235          SBLKX:  DS     2
B237          SIZEX:  DS     2

B239                  END    START
```

How to Merge 'BASIC' Programs with Assembly Language Programs

Rick Taubold
197 Hollybrook Road
Rochester, NY 14623

by Rick Taubold (and Tom Devlin, who helped but wants none of the credit)

Let me ask all of you a question. How many of you have seen one of those programs which you could LIST in BASIC but which obviously contained more? Perhaps there was a CALL instruction but no machine language had been loaded either from disk or by POKEing. How many raised hands do I see? The purpose of this article is to clear up this little mystery. In the process you will learn new things about your Compucolor II. What is described here is not limited to the CCII, but will work on any computer that employs Microsoft or similar BASIC.

It was Tom Devlin, maker of nifty hardware for the Compucolor II, who first shared the secret with me. I should also point out that this can be used to merge any number of machine language subroutines with one BASIC program as well as permitting you to SAVE a machine language program as if it were a BASIC program. BASIC is a flexible language. Unfortunately, it is occasionally too slow for all desired uses. Writing entire assembly language programs might be fun to some people. To most of us it's a lot of work. Therefore, it is often desirable to write in BASIC and to add short machine language routines where speed is required. The CALL function in BASIC allows interfacing to machine language subroutines. Since it involves a subroutine, it must always end with the machine language equivalent of a RETURN instruction (RET in mnemonic code, hex value = C9, decimal value = 201). Other times it is convenient to write most of the program in machine language but to write an introduction or instructions in BASIC.

In this case there are several options. Usually the programmer will simply use BASIC to load and run his machine language program directly in which case there is no difficulty. An alternative is to load both programs at the same time and to employ ESC USER to execute the machine language. Again, the programs exist separately on the disk. I will present ways of having both BASIC and machine language programs in memory together but merged as a single program on the disk. Interested? Read on.

Before I continue, I would like to clear up a few misconceptions about the CALL and ESC USER functions of the CCII. Many users seem to think that ESC USER is limited to a single function. This is untrue. Both the CALL and ESC USER commands represent what is called a 'JUMP VECTOR'. By way of explanation let's consider an analogy in BASIC. Assume that we have a command like GOTO X or GOSUB X, where X could be a variable instead of a specific line number. Wouldn't the capability be nice? Our variable X would be a set variable names but we could change the value of X whenever we wished.

A JUMP VECTOR is similar. It means that we are telling the computer to jump to a particular fixed location. At that location is another jump instruction. The CALL command uses the locations 33282, 33283 and 33284. ESC USER employs the three locations 33215, 33216, 33217. The first location (33282 or 33215) always contains a machine language JMP instruction (C3 hex, 195 decimal) which is similar to the GOTO and GOSUB instructions in BASIC. This instruction is followed by a 2-byte memory address. Most of the time the programmer only uses one jump address. A typical program will assign only one CALL function. However, this is not a requirement. In my 'FINAL FRONTIER' program several different machine language routines are used, depending on the need at the time. Since most of the program was written in BASIC and only one CALL command available, the 'jump vector' must be POKEd with a new jump address each time a different machine language subroutine is required.

The CALL command in BASIC operates as a GOSUB to a machine language subroutine. When the machine language subroutine is completed, the program will RETURN to the next BASIC statement. When the BASIC sees a CALL command, it immediately jumps to memory location 33282 and sees another JMP command. (Keep in mind that ANY instruction could be placed here.) The BASIC reset routine normally insures that a JMP command is placed here, but I like to POKE in the command just in case it is somehow wiped out along the way. The computer will execute whatever it sees, so it pays to be certain which command is there.

The jump address is calculated in an unusual way (unusual only if you're not used to it). As an example, let us assume that the machine language subroutine to be CALLed is at hex location F000 (61440 decimal). From this we must calculate two values to POKE: F0 and 00. These work out to be, in decimal, 240 and 0 respectively. However, when machine language reads an address, it expects the two bytes IN REVERSE ORDER! Therefore, we POKE them in backwards, and our final POKE instruction line to set up this particular CALL would be:

```
POKE 33282,195:POKE 33283,0:POKE 33284,240
            (JMP)          (00)          (F0)
```

When we use X = CALL(0) from BASIC, our program will first jump to location 33282, see the JMP F000 instruction, and go to F000 hex to begin execution.

The ESC USER works the same way. The only difference is that ESC USER acts like a GOTO and recognizes no RETURN instruction. Again, the important thing to remember is that we are not restricted to a single jump location. Your program can alter these jump vectors at any time,

and this makes them extremely powerful. ESC USER can be executed directly from a BASIC program by PLOT 27,30.

There are several other places where jump vectors are used in the CCII. One of these is USER TIMER -2, and another is INPCRT. If anyone out there is interested, I can cover these in a future article. Now let us return to our main topic. You'll see the relevance of the previous discussion shortly.

I will demonstrate the merging procedure using the Scrolling Patch, a simple but useful illustration. In its original form the Scrolling Patch used a BASIC program of some 500 bytes to POKE in a 32 byte machine language program. Somehow, this seems like overkill. For many applications the programmer can enter this Patch directly and 'throw away' the BASIC part. New parameters can be POKEd easily. This method saves memory and cleans up a program. A more or less complete description of this Patch appeared in the double issue Nov/Dec-Jan/Feb of FORUM. The first step in the procedure is to write and test the assembly language program. I have already done this in Listing -2. When everything works, you are ready to merge the two. For this demonstration enter the BASIC program in Listing -1, exactly as written, and SAVE it on disk. Next, using a screen or text editor, enter the source code of Listing -2 and save it on disk also. You may omit the comments. I placed the scroll parameters in EQU statements so that you can readily change them for your purposes. Do not assemble the source code yet!

You should now have the two key programs on disk. So far, nothing out of the ordinary has been done. Two methods of merging are presented, each having its own advantages and disadvantages.

RICK'S METHOD:

This method yields the most compact program but requires that you change and reassemble the source code whenever you change the length of the BASIC program it is to be used with. It assumes that the machine code will begin immediately after the BASIC code. In addition, any changes will require that you also change the CALL jump vector. When you write the BASIC program, you never know until you're done exactly where it will end. When you set up the POKEs for the CALL vector, the values can be 1, 2 or 3 digits long. You appear end up in a no-win situation. If the number of digits changes, the length of the BASIC program changes which in turn changes the CALL vector which means changing the BASIC program, and so on...

Simply make all 3 numbers three digits long. The first POKE will always be 195. Make the other two both 000. In this way you can change the numbers without changing the program length. BASIC won't care if you POKE 33283,019 instead of POKE 33283,19.

Before you can assemble the source code you must know the ORG address, that is, where it will be loaded. This address will become the same as the end address of the BASIC program. If you have 'The' BASIC EDITOR, load the BASIC program and note the END@ number (in hex) at the bottom of the screen. Otherwise, you can get this value from the disk directory. It's in the SADR column of the directory. This hex address now becomes your ORG address. It should be 8384 if you typed the program as written (watch the spacing in the REM). Use your screen editor to change the ORG then assemble the program. If you are using the original CCII assembler (as opposed to the Macro Assembler, which is frequently more trouble than it's worth), you can leave the file .LDA. There is no need to convert to .PRG. Note the address of the last assembled instruction at the ENDPRG label which the assembler prints out when it's done. This should be 83A4. You'll need this in a moment.

LOAD the BASIC program. From FCS, LOAD your machine language program. The two programs are now back to back in memory. Here's where the trick comes in. In order to SAVE the whole mess from BASIC, we need to tell BASIC where the new program ends. The only end address it currently has is the old one of the BASIC program. However, we've extended it by adding the machine language portion..

Now you need that last address at the ENDPRG label in your assembly printout. In the Programming Manual one of the 'Key Memory' locations listed is 32982 (Points to end of BASIC source and start of BASIC variables). This and 32983 are the locations which we must change to fool BASIC. The start of variables pointer (SOV) marks the end of the actual BASIC program and the start of the memory area where BASIC's variables can start. This location changes every time you add to or delete lines in the program. that's why a pointer is needed, so we don't waste space. The variables start right after the program ends. By using this pointer location we are fooling BASIC into thinking that our machine language program is part of the BASIC program. This protects the routine so it cannot be wiped out by variables, etc. When using this pointer remember to POKE the A4 first (164 decimal) then the 83 (131 decimal). Use IMMEDIATE MODE and type:

```
POKE 32982,164:POKE 32983,131 <RETURN>
```

We're all set. Simply, SAVE the program from BASIC. It can be LOADed from BASIC and RUN as any other program. Test it. Just remember that if you make even the tiniest change in the BASIC program, you'll have to remerge the two using a new value for the ORG address. This difficulty is overcome by--

TOM'S METHOD:

This procedure requires that you assemble the source code twice. It also yields a somewhat longer total program, although this may be inconsequential. The big advantage is that you can make minor changes to the BASIC program without having to reassemble the source code. For most applications, this will be the better method. First, change the source code in Listing 2 as follows (A.L. stands for assembly language):

After the instruction W EQU 30 add—

```
ENDAL    EQU    849AH    ;WHERE WE WANT A.L. TO END
```

Between the ENDPRG label and END START add—

```
ENDPRG: REORG  EQU     ENDAL-($-START)
        ORG    32982   ;START OF VARIABLES POINTER
        DW     ENDPRG  ;MOVE POINTER TO PROTECT A.L.

        END    START
```

In this procedure the main difference is the start address of the machine language program. One advantage is that we can make the total program exactly fill a given number of disk sectors. (One disk sector holds 128 bytes.) Because BASIC begins at 829A (hex), additional numbers ending in 001A hex (e.g. 831A) will fill an odd number of disk sectors and those ending 009A hex (e.g. 839A) will fill an even number of disk sectors. As an example, we chose 4 sectors, making the end address 849A hex. This will add sufficient 'space' between BASIC and the assembly language for future changes.

The next trick is to discover the ORG address for the assembly language. We want it to end at 849A, but, until we assemble the program, we don't know how long it will be. The first line after the ENDPRG label helps us to accomplish our goal. ENDAL is set at the start as 849A. In the expression the '$' symbol is a notation for the current assembler address. By subtracting the address of START from it, we get the difference, or the length of the program. Subtracting this result from the address ENDAL, we calculate the starting address of the machine language. Now assemble the program for the first time. The assembler will print the REORG address (assuming you look for it) in parentheses. It should come out as 847A. Go back and change the initial ORG with the editor from 8384 to 847A. Reassemble the edited program. If you did everything right, REORG should come out the same as ORG. All that remains to be done is to change the CALL vectors in the BASIC program to reflect the new location of the machine language and to merge the two programs, as with Rick's method. To change the CALL vector, line 120 of the BASIC program becomes:

```
POKE 33282,195:POKE 33283,122:POKE 33284,132
```

With Rick's method you had to manually POKE the pointer values at 32982 & 32983. With Tom's method, we let the assembler do it for us. By setting the second ORG at the end to 32982 and using the DW (define word, 2 bytes) directive, we can insert the end address (ENDPRG label) into the required memory locations. A word of caution is in order. You must use the old assembler's .LDA file to do this. The .PRG file won't work! If you must create a .PRG file, you will have to POKE 32982 and 32983 manually as with Rick's method. In either case, you can still SAVE the entire program from BASIC.

Before concluding, I need to mention a couple of possible bugs. The first one is that you cannot LOAD these hybrid programs using the DOS of 'The' BASIC EDITOR. Apparently this editor uses a different method to calculate the end of the BASIC program rather than using the SADR address on disk. The other possible problem is that using 'The' BASIC EDITOR's HELP feature will effectively strip off the assembly language program when you attempt to reSAVE it. Therefore, don't take chances. Use BASIC's direct SAVE and LOAD commands and everything will be fine. □

LISTING #1

```
100 REM TEST OF SCROLL PATCH
110 PLOT 12,15
120 POKE 33282,195:POKE 33283,132:POKE 33284,131
130 LN=9
140 FOR J=1 TO 40
150 IF LN<19 THEN LN=LN+1:PLOT 3,10,LN:GOTO 180
160 X=CALL(0)
170 PLOT 3,10,LN:PRINT SPC(30)"":PLOT 3,10,LN
180 PLOT 6,J:PRINT J;" TESTING---SCROLLING---"
190 NEXT J
200 PLOT 8,6,2
210 END
```

LISTING #2 (Rick's version)

```
;SCROLL PATCH ADD ON TO 'BASIC' PROGRAM

;SCROLL AREA PARAMETERS..................

X       EQU     10      ;STARTING COLUMN ON SCREEN
Y       EQU     10      ;STARTING ROW ON SCREEN
H       EQU     10      ;# OF LINES TO SCROLL
W       EQU     30      ;# OF CHARACTERS WIDE TO SCROLL

        ORG     8384H   ;END ADDRESS OF 'BASIC' PROGRAM

        ;28672 is start of screen memory (X=0, Y=0)
        ; so first line below is starting screen location

START:  LXI H, 28672+128*Y+X+X
        MVI B, H-1          ;count lines

LOOP2:  MVI C, W*2          ;how wide before next line

LOOP1:  LXI D, 0080H        ;128 decimal (down 1 line)
        DAD D
        MOV A,M             ;get a byte
        LXI D, 0FF80H       ;-128 decimal (back up 1 line)
        DAD D
        MOV M,A             ;reload byte in new location
        INX H               ;next location
        NOP                 ;becomes INX H w/no color scroll
        DCR C               ;done with this line?
        JNZ LOOP1           ;no
        LXI D, 128-W-W      ;yes, next line
        DAD D
        DCR B               ;done all lines?
        JNZ LOOP2           ;no
        RET                 ;yes, back to BASIC program

ENDPRG:

        END     START
```

A PASCAL FOR THE COMPUCOLOR II

Part III. A Roadmap to successful installation and use.

Doug Van Putte
18 Cross Bow Drive
Rochester, NY 14624

In this part of the Tiny-Pascal series we will provide step-by-step instructions for installing Tiny-Pascal, hoping to provide more readers with enough inspiration to tackle this tutorial series. While the necessary documentation was referenced in Part I, the method of installation can be intimidating to those willing but unfamiliar with the approach used.

The installation instructions are taken from the implementation of Tiny-Pascal, written in the FORTH language, prepared by Dr. Jim Minor. Both the FORTH language and the Tiny-Pascal language are utilized in this installation, and both are available from the CHIP library by writing to the author. FORTH is supplied as a PRG file, while Tiny-Pascal is supplied in the form of FORTH programs, or 'screens.'

FORTH (and Tiny-Pascal) utilizes the disk in 1024 byte units called 'screens.' The screen contains the program code lines, and is used as the means of displaying, entering, and editing programs using a special editor. A screen consists of 16 lines of 54 characters. Only one screen can be displayed on the CRT at a time. A program may consist of more than just one screen, each screen linked to the other by a special coding. A disk side can hold fifty screens, numbered 0 to 49. Blank screens, or screen templates, are used to enter new programs, so it is convenient to have a 'starter screen set' from which to begin.

A starter set of screens for FORTH is supplied with contents on screens 0-19, starting at block 0. Screens 0 and 1 contain a conventional, but dummy, FCS directory. Screens 2 and 3 contain 'boilerplate.' Screens 4 and 5 contain compiler error messages. Screens 6 to 16 contain an editor which will be bypassed in favor of a better editor that comes as part of the Tiny-Pascal disk. Special FORTH words are contained on screens 17 to 19. I recommend that these FORTH starter sets be backed up by using a disk copy program that works without a directory. This copy should then be used as a 'program' disk, to hold programs written on blank screen templates.

The starter set of screens for Tiny-Pascal is supplied with contents on screens 0-9 and 20-37. Screens 0-1 and 4-5 are utilized for the same purpose as those on the FORTH starter set, ie: FCS directory and error messages. Screens 2 and 3 contain the Tiny-Pascal error messages, and 6-9 contain sample Tiny-Pascal programs. Blank templates for program development are provided on screens 10 to 19. Screens 20-37 contain a FORTH line editor which is superior to the editor provided on the FORTH disk. This is the editor we are using to enter and modify code. The Tiny-Pascal set should also be backed up before program development begins.

Our objective is to install Tiny-Pascal (the compiler) and to enter a Tiny-Pascal program using the screen editor. Once the Tiny-Pascal compiler has itself been compiled and saved in PRG format, subsequent sessions with Tiny-Pascal are significantly simplified. [This process consists of saving an expanded version of FORTH which contains Tiny-Pascal command words. With such a version, the FORTH responds to Tiny-Pascal commands as though it were a Tiny-Pascal system. Ed.] You will need CHIP FORTH Disk #46, and the two Tiny-Pascal disks, No. 83 and No. 84. In the text below, '(cr)' means press the carriage return key.

1) Place the FORTH disk, CHIP #46, in the disk drive. From FCS type **RUN FORTH(cr)**

You should see the FORTH prompt 'OK.' If it does not appear, press (cr).

2) Type at the FORTH prompt,

HEX 1 1A +ORIGIN ! COLD(cr)

This will turn on the error message text.

3) Place CHIP Disk - 83 in the disk drive. Type

6 LOAD(cr)

This will compile Tiny-Pascal into FORTH. Wait patiently.

4) Now place a fresh formatted disk into the drive. Type

SAVE TPAL4(cr)

This will save the augmented FORTH, containing Tiny-Pascal commands, to disk.

5) Place CHIP Disk # 84 into the drive. Type

20 LOAD(cr)

This will compile the line editor into our augmented version of FORTH (TPAL). Wait patiently again.

6) Replace the previously-used formatted disk in the drive. Type

SAVE TPALED(cr)

This will save our FORTH/PASCAL Editor to disk.

Having made these changes, we need no longer be concerned with them when using Tiny-Pascal. Our two Tiny-Pascal programs, TPAL4.PRG and TPALED.PRG, can be loaded from FCS using the RUN command from now on. These two programs are our Tiny-Pascal system.

Assume, now, that we wish to create a new program on a blank screen template and save it to disk.

7) Place the Tiny-Pascal system disk, containing TPAL4 and TPALED, into the disk drive. Type

RUN TPALED(cr)

This will load the line editor. Wait for the 'OK' prompt.

8) Now place your backup copy of Disk #84 into the drive. Type

10 LIST(cr)

This will load a blank screen template.

9) To invoke the line editor, type **EDITOR<cr>**

10) You may now type in a program. Part I of this series lists the editor commands. Refer to them for assistance. You may use the sample program given in Part II of this series for practice.

11) When the sample program is all typed in, type

```
FLUSH<cr>
```

to save screen #10 just entered.

12) To invoke the Pascal compiler, type

```
PASCAL<cr>
```

13) To compile the program you just wrote on screen #10, type

```
10 LOAD<cr>
```

14) To run the program just compiled, type just the program name at the prompt. For example, to run Rectanglearea, just type

```
RECTANGLEAREA<cr>
```

at the prompt.

The following commands may be used for editing a program. You will want to delete the compiled program, containing the errors, before compiling a new one of the same name.

14) To delete a compiled program, you may use the FORTH 'FORGET' command. For example, to delete a bad version of RECTANGLEAREA, type

```
FORTH FORGET RECTANGLEAREA<cr>
```

before re-compiling an edited program version. This will delete the 'bad' version from memory.

```
FORTH FORGET <filename><cr>
```

will delete any compiled program.

15) To re-invoke the screen editor for correcting mistakes or any editing of a program, type

```
EDITOR<cr>
```

16) To get screen #10 back again, type

```
10 LOAD<cr>
```

Now the text may be edited using the line editor commands. You will need to recompile following steps 11, 12 and 13 above.

What if your entire program takes more than one screen? There is a FORTH word (a symbol, really) that 'tells' the compiler there is yet another screen connected with this program. This word is the 'continuation command' and looks like this: **-->**

It is always preceeded by a space. The continuation command is entered after the last Pascal command on the screen. The continuation command assumes that the 'continuation' is on the screen with the next higher number from the screen on which it appears. You cannot, for example, continue screen #15 on screen #18. Screen #15 must continue on screen #16.

How do you get more blank screens? Use the Tiny-Pascal command(really a FORTH command we are borrowing) COPY to create blank screens. Assume screen #19 is blank, and screen #20 contains information you can erase.

17) Load TPALED by typing from FCS,

```
RUN TPALED<cr>
```

18) To invoke the line editor, type

```
EDITOR<cr>
```

19) To make a copy of screen #19 on screen #20, type

```
19 20 COPY<cr>
```

This overwrites the contents of screen #20 with a blank template.

Or you may do it another way. First perform steps 17 and 18 above.

20) Type **20 LIST<Cr>**

This will load the screen we want to blank out.

21) To clear screen #20 in memory, type **WIPE<cr>**

22) When you have entered the new program lines, type

```
FLUSH<cr>
```

to save the new screen #20 to disk.

It is not advisable to blank screens 0-5 on the disk, since they contain the directory and the error messages. Any screen beyond 5 may be used for programs. Preserve the original sampler disks, however. Use backup disks for your programs.

You may not change the screen number of any screen, but the contents of a screen may be moved to another screen using the COPY function from step 19, above. The source screen may then be blanked if desired.

How can you remember which screen a program is on? You can keep a logbook of screen contents, or the 'dummy directory' may be used to 'log' the screens from 6 to 49. Lets add a directory entry for RECTANGLEAREA. Assume this program is on screen #10.

23) Place the Tiny-Pascal system disk in the drive.

24) From FCS type,

```
SAVE RECTAR.TPL 0 10<cr>
```

This command makes a directory entry consisting of the program name, RECTAR, and the number 10, which references the originating screen of the program. The number of entries in your 'program catalog' (FCS directory) is limited to 13. This 'catalog' of programs can be listed directly from FCS with the DIR command.

Now you have all the resources to get started with Tiny-Pascal. Gather together your FORTH and Tiny-Pascal disks, follow the course provided, and journey to a fine experience. Best wishes on a safe trip! □

UNCLASSIFIED ADVERTISEMENTS

Forum - back issues still available!

RAM/EPROM for the CCII

Back issues of COLORCUE contain a wealth of practical information for the beginner as well as the more advanced programmer, and an historical perspective on the CCII computer. Issues are available from October 1978 to current.

DISCOUNT: For orders of 10 or more items, subtract 25 % from total after postage has been added. **POSTAGE:** for U.S., Canada and Mexico First Class postage is included; Europe and South America add $1.00 per item for Air Mail, or $ 0.40 per item for surface; Asia, Africa, and the Middle East add $ 1.40 per item for Air Mail, or $ 0.60 per item for surface. **SEND ORDER** to Ben Barlow, 161 Brookside Drive, Rochester. NY 14618 for VOL I through VOL V; and to Colorcue, 19 West Second Street, Moorestown, NJ 08057 for VOL VI and beyond.

1978	**VOL I**	**$3.50 each**
	No. 1-3:	OCT/ NOV/ DEC
1979	**VOL II**	**$3.50 each**
	No. 1-3:	APR/MAY/JUN
	No. 4-5:	JAN/FEB/MAR
	No. 6-7:	AUG/SEP/OCT
	No. 8:	NOV XEROX COPY, $2.00
1980	**VOL III**	**$1.50 each**
	No. 1	DEC/JAN
	No. 2:	FEB
	No. 3:	MAR
	No. 4:	APR
	No. 5:	MAY
	No. 6:	JUN/JUL
1981	**VOL IV**	**$2.50 each**
	No. 0:	DEC/JAN
	No. 1:	AUG/SEP
	No. 2:	OCT/NOV
1982	No. 3:	DEC/JAN
	No. 4:	FEB/MAR
	No. 5:	APR/MAY
	No. 6:	JUN/JUL
	VOL V	
	No. 1:	AUG/SEP
	No. 2:	OCT/NOV
1983	No. 3:	DEC/JAN
	No. 4:	FEB/MAR
	No. 5:	APR/MAY
	No. 6:	JUN/JUL
1984	**VOL VI**	**$3.50 each**

-----DISK VOLUME SPACE-----

ADDR	BYTE	TRANS	FUNCTION
8200			
01			Marker
02			Attribute
03			
04			
05			
06			Volume
07			Name
08			
09			
0A			
0B			
0C			
0D			
0E			
0F			
8210			Not
11			Used
12			
13			
14			
15			
16			

-------1ST FILE ENTRY-------

ADDR	BYTE	TRANS	FUNCTION
8217			Attribute
18			
19			
1A			File
1B			Name
1C			
1D			
1E			File
1F			Type
8220			
21			Version
22			SBLK
23			0005
24			SIZE
25			0005
26			LBC
27			LADR
28			8200
29			SADR
2A			
2B			Spare

-------2ND FILE ENTRY-------

ADDR	BYTE	TRANS	FUNCTION
822C			Attribute
2D			
2E			
2F			File
8230			Name
31			
32			
33			File
34			Type
35			
36			Version
37			SBLK
38			000A
39			SIZE
3A			1202
3B			LBC
3C			LADR
3D			
3E			SADR
3F			
8240			Spare

-------3RD FILE ENTRY-------

ADDR	BYTE	TRANS	FUNCTION
8241			Attribute
42			
43			
44			File
45			Name
46			
47			
48			File
49			Type
4A			
4B			Version
4C			SBLK
4D			
4E			SIZE
4F			
8250			LBC
51			LADR
52			
53			SADR
54			
55			Spare

-------4TH FILE ENTRY-------

ADDR	BYTE	TRANS	FUNCTION
8256			Attribute
57			
58			
59			File
5A			Name
5B			
5C			
5D			File
5E			Type
5F			
8260			Version
61			SBLK
62			0005
63			SIZE
64			0005
65			LBC
66			LADR
67			8200
68			SADR
69			8200
6A			Spare

-------5TH FILE ENTRY-------

ADDR	BYTE	TRANS	FUNCTION
826B			Attribute
6C			
6D			
6E			File
6F			Name
8270			
71			
72			File
73			Type
74			
75			Version
76			SBLK
77			000A
78			SIZE
79			1202
7A			LBC
7B			LADR
7C			
7D			SADR
7E			
7F			Spare